

Escuela Politécnica Superior

19
20

Trabajo fin de grado

NIDS basado en Aprendizaje Automático



José Ignacio Gómez García

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C/ Francisco Tomás y Valiente nº 11

**UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR**



Grado en Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

NIDS basado en Aprendizaje Automático

**Autor: José Ignacio Gómez García
Tutor: Carlos Alaíz**

junio 2020

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución comunicación pública y transformación de esta obra sin contar con la autorización de los titulares de la propiedad intelectual.

La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. del Código Penal*).

DERECHOS RESERVADOS

© 10 de Junio de 2020 por UNIVERSIDAD AUTÓNOMA DE MADRID

Francisco Tomás y Valiente, n^o 1

Madrid, 28049

Spain

José Ignacio Gómez García

NIDS basado en Aprendizaje Automático

José Ignacio Gómez García

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A Paloma. Porque sé que te habría hecho ilusión.

El destino suele estar a la vuelta de la esquina. Como si fuese un chorizo, una furchia o un vendedor de lotería: sus tres encarnaciones más socorridas. Pero lo que no hace es visitas a domicilio. Hay que ir a por él.

La Sombra del Viento, Carlos Ruiz Zafón

AGRADECIMIENTOS

En primer lugar, me gustaría agradecer a mi tutor, Carlos Alaíz, su labor de orientación a lo largo de este proyecto.

También me gustaría dar las gracias a mi compañeros y amigos, Alejandro Cabana y Héctor Palencia, los consejos aportados y su ayuda como conejillos de indias de la aplicación.

Especial mención a Ana Triguero por leer, releer y volver a leer este documento, luchando fervientemente contra mis brotes de dislexia.

Finalmente, gracias a mis padres, José Ignacio y Begoña, por aguantar mis días de agobio (especialmente al final) y apoyarme durante las largas jornadas delante del monitor en plena pandemia.

RESUMEN

Hoy en día, en plena era del Internet of Things (IoT), cuando la cantidad de dispositivos conectados en nuestros hogares no hace más que aumentar, la Ciberseguridad debe ser considerada como una de las disciplinas más importantes para nuestra sociedad.

Esto es algo que las grandes empresas tienen muy claro, y ya cuentan con departamentos enteros dedicados a gestionar la seguridad en sus redes, dispositivos y productos. Además, existe una gran cantidad de soluciones especializadas (algunas de código abierto) como antivirus, *firewalls*, Sistemas de Detección y Prevención de Intrusiones, etc.

Sin embargo, en un entorno doméstico, la Ciberseguridad sigue siendo una de las principales tareas pendientes.

El objetivo de este proyecto es el de desarrollar una aplicación de código abierto, llamada VISION, capaz de detectar ataques en una red doméstica y dotar al usuario de las herramientas necesarias para mitigarlo inmediatamente.

VISION es un Sistema de Detección de Intrusiones en Redes sencillo, modular y escalable que integra un modelo de detección de ataques que puede ser entrenado usando algoritmos de Machine Learning sobre datos de la propia red en la que vaya a ser desplegado. Cuenta con una interfaz de control que permite al usuario monitorizar la red y reaccionar ante cualquier ataque, bloqueando la fuente del mismo.

En este documento se incluyen los resultados obtenidos al entrenar modelos de detección de ataques sobre diversos *datasets* y se profundiza en el diseño y el desarrollo de VISION, para presentar la primera versión de esta herramienta de código abierto.

PALABRAS CLAVE

Aprendizaje Automático, Sistema de Detección de Intrusiones, Ciberseguridad, Monitor de Tráfico

ABSTRACT

Nowadays, in the midst of the IoT (Internet of Things) Revolution, while the amount of connected devices at our homes keeps growing, Cybersecurity must become one of the most important issues for our Society.

Most companies have a clear mind about that, so they already have entire departments focused on keeping their networks, devices and products protected. Moreover, there is an increasing amount of professional solutions aiming to fortify their defenses, like antivirus, firewalls, Intrusion Detection and Prevention Systems, etc.

However, when we talk about our own homes, Cybersecurity is always a step behind and has become one of our pending tasks.

This project aims to develop an open source application, called VISION, capable of detecting attacks inside a home network and giving the user tools to mitigate its effects.

VISION is a modular, scalable Network Intrusion Detection System which includes a Machine Learning-based attack detection model. It can be trained using data from the same network where it will be deployed and has a control interface designed to give the user the ability to react against any attack, blocking its source.

This document includes the results obtained after training some models over different datasets. It also delves into VISION design and development stages showing a full first version of this open source tool.

KEYWORDS

Machine Learning, Intrusion Detection System, Cybersecurity, Traffic Monitor

ÍNDICE

1	Introducción	1
1.1	Motivación	1
1.2	Objetivos	2
1.3	Estructura del documento	2
1.4	Contenido del proyecto	4
2	Sistemas de Detección de Intrusiones	5
2.1	Trabajo relacionado	5
2.1.1	Snort	6
2.1.2	Hogzilla	6
2.1.3	Suricata	6
2.1.4	SolarWinds	6
2.2	Ataques en redes	6
2.2.1	Denegación de Servicio (DoS)	6
2.2.2	Escaneo de puertos	7
2.2.3	Ataque por fuerza bruta	7
2.3	Aportación del proyecto	7
3	Modelo de detección automática de intrusiones	9
3.1	Dataset CSE-CIC-IDS2018	9
3.1.1	CICFlowmeter	11
3.2	Preprocesamiento	11
3.3	Entrenamiento	13
3.3.1	Regresión Logística	13
3.3.2	Hiperparámetros	14
3.3.3	Entrenamiento y Test	15
3.3.4	Estandarización	15
3.3.5	Implementación	15
3.4	Dataset VISION-IDS2020	16
3.4.1	Infraestructura	17
3.4.2	Metodología	18
3.4.3	Resultado	19
4	Resultados experimentales	21

4.1 CSE-CIC-IDS2018	21
4.1.1 Conclusión	23
4.2 VISION-IDS2020	25
4.2.1 Conclusión	27
5 Diseño de VISION	29
5.1 Módulos	29
5.1.1 Módulo <i>Backend</i>	30
5.1.2 Módulo <i>Frontend</i>	30
5.2 Requisitos de la aplicación	31
5.2.1 Requisitos funcionales	31
5.2.2 Requisitos no funcionales	31
5.3 Clases	32
5.4 Arquitectura	33
6 Desarrollo e implementación de VISION	35
6.1 Infraestructura	35
6.1.1 NIDS	37
6.1.2 Database	37
6.1.3 Frontend	37
6.2 Estructura de ficheros	38
6.3 API	38
6.4 Escalabilidad	38
6.5 Seguridad	38
7 Conclusiones y trabajo futuro	39
7.1 Conclusiones	39
7.2 Trabajo Futuro	40
Bibliografía	41
Definiciones	43
Acrónimos	45
Apéndices	47
A Características utilizadas para el modelo	49
B Requisitos de la aplicación	53
B.1 Requisitos Funcionales	53
C Tecnología empleada	61
C.1 NodeJS	61

C.2	VueJS	62
C.3	Python	62
C.4	MySQL	62
C.5	Sequelize	62
C.6	Docker	63
D	Estructura de ficheros	65
E	Métodos de la API	67
E.1	Métodos de la API NIDS	67
E.2	Métodos de la API Frontend	68
F	Ejemplos de escalabilidad de la aplicación	69
F.1	Utilizar un nuevo modelo de predicción	69
F.2	Añadir una nueva gráfica al Frontend	69
G	Seguridad	71
G.1	Seguridad en Docker	71
G.2	Tráfico cifrado	71
G.3	Usuario único	72
G.4	Seguridad en la API	72
G.5	Gestión segura de contraseñas	74
G.6	Bloqueo de IPs	74
H	Manual de usuario	77
H.1	Instalación y despliegue	77
H.2	Manual de uso	77
H.2.1	Primer inicio de sesión	78
H.2.2	Inicio de sesión	78
H.2.3	Ventana principal	78
H.2.4	Lanzamiento del sniffer	78
H.2.5	Dashboard	80
H.2.6	Gráficas	81
H.2.7	Objetivos	81
I	IPTables	83
I.1	Composición de IPtables	83
I.2	Generación de reglas	84
I.3	Limitaciones de la aplicación	84

LISTAS

Lista de códigos

3.1	Código para recortar el <i>dataset</i>	13
3.2	Entrenamiento del modelo.	16

Lista de figuras

3.1	Tabla de ataques CIC	10
3.2	Proporción benignos/ataques en Sample1	12
3.3	Proporción entre paquetes benignos y ataques en la muestra Sample2.	12
3.4	Infraestructura Raspberry	17
3.5	Figura que muestra la proporción entre tráfico benigno y cada tipo de ataque en el <i>dataset</i> generado.	19
4.1	Curva ROC del modelo entrenado con CIC.	22
4.2	Curva ROC CIC vs. Real.	23
4.3	Comparación entre datasets.	24
4.4	Resultado de LogReg2 usando un día aleatorio del CIC.	24
4.5	Curva ROC modelo VISION.	26
4.6	Curva ROC modelo VISION sobre <i>dataset</i> de test.	27
4.7	Curva ROC modelo VISION sobre <i>dataset</i> CIC.	28
5.1	Módulos de la aplicación.	29
5.2	Requisitos no funcionales.	32
5.3	Diagrama de clases.	33
6.1	Capas de la aplicación.	35
6.2	Diagrama de contenedores Docker.	36
B.1	RF1.	53
B.2	RF2.	54
B.3	RF3.	55
B.4	RF4.	55
B.5	RF5.	56
B.6	RF6.	57

B.7	RF7.	58
B.8	RF8.	58
B.9	RF9.	59
G.1	Certificado SSL.	72
G.2	Obtención de tokens.	73
H.1	New Login.	78
H.2	Login.	79
H.3	Login.	79
H.4	Menu superior.	79
H.5	Lanzamiento del sniffer.	80
H.6	Dashboard.	80
H.7	Gráficas.	81
H.8	Fijar objetivos.	82
H.9	Tabla de objetivos.	82
H.10	IPs más atacadas por un objetivo.	82
I.1	Esquema de acción de las distintas tablas en <i>iptables</i>	83

Lista de tablas

3.1	Franjas horarias de los ataques realizados para nuestro <i>dataset</i>	19
4.1	Modelos entrenados con el <i>dataset</i> CSE-CIC-IDS2018.	21
4.2	Matriz de confusión del modelo LogReg2 entrenado sobre el CSE-CIC-IDS2018.	22
4.3	Report del modelo LogReg2 entrenado sobre el CSE-CIC-IDS2018. Refleja los resultados obtenidos sobre el conjunto de test.	22
4.4	Matriz de confusión al usar como conjunto de entrenamiento una muestra real.	23
4.5	Modelos entrenados con el <i>dataset</i> VISION-IDS2020.	25
4.6	Matriz de confusión del modelo entrenado sobre el VISION-IDS2020.	25
4.7	Report del modelo entrenado sobre el VISION-IDS2020.	26
4.8	Matriz de confusión del modelo entrenado sobre el VISION-IDS2020, sobre el conjunto VISION-IDS2020-TEST.	27
4.9	Matriz de confusión del modelo entrenado sobre el VISION-IDS2020, usando como conjunto de test el CSE-CIC-IDS2018.	28
A.1	Lista de features extraídos por CICFlowmeter.	51
E.1	Lista de métodos de la API NIDS.	68

E.2	Lista de métodos de la API Frontend.	68
-----	---	----

INTRODUCCIÓN

En este primer capítulo se expone la motivación para el proyecto, así como sus objetivos principales y la estructura del documento.

1.1. Motivación

Un Network Intrusion Detection System (NIDS) o, en castellano, Sistema de Detección de Intrusiones en Redes, es una herramienta capaz de detectar intentos de acceso no autorizados a una red.

Hoy en día, la mayoría de los NIDS basan su funcionamiento en la aplicación de heurísticas o reglas estáticas que surgen a raíz de que un analista detecte patrones en ataques previos. Sin embargo, muy pocos sistemas incluyen actualmente técnicas de aprendizaje automático para detectar esos patrones.

Esto puede deberse, en gran parte, a que se necesita un conjunto de datos bastante extenso para poder entrenar un modelo de forma eficaz, y existe una gran reticencia por parte de las empresas para compartir estos datos, puesto que en muchos casos pueden contener información sensible.

La idea de este proyecto surge a raíz de descubrir la existencia de un *dataset* generado por el Canadian Institute for Cybersecurity (CIC), denominado **CSE-CIC-IDS2018** [1]. Se trata de un *dataset* con una extensión notable (entorno a 100 GB) que incluye muestras de varios ataques comunes y conocidos.

Decidimos entonces diseñar una aplicación, que bautizamos como **VISION**, capaz de detectar ataques en una red y notificarlos de forma que un analista pueda actuar en consecuencia en tiempo real.

Uno de los puntos de partida del proyecto fue el artículo publicado por Tamim Mirza en el blog *Towards Data Science* [2], en el que presenta su propio Trabajo de Fin de Grado centrado en la obtención de un modelo de *Deep Learning* para la detección de ataques sobre una versión previa de este *dataset*. Como se describirá más adelante, se decidió utilizar algoritmos de aprendizaje automático más sencillos para poder dedicar esfuerzo también al desarrollo de la aplicación. Estos algoritmos presentan un rendimiento más que satisfactorio para una primera aproximación al problema.

1.2. Objetivos

Este proyecto tiene dos objetivos principales. Por un lado, se pretende analizar el *dataset* CSE-CIC-IDS2018 y obtener un modelo capaz de detectar ataques a partir de un conjunto de datos extraídos de una captura de red. El otro objetivo es el de desarrollar una aplicación que permita a un usuario saber en todo momento si se está produciendo algún ataque y, en ese caso, mitigarlo lo antes posible.

Se establecieron, por tanto, los siguientes hitos para el proyecto:

- 1.– Analizar el *dataset* CIC-CSE-IDS2018.
- 2.– Tratar de obtener un modelo preciso a partir del CIC-CSE-IDS2018.
- 3.– Analizar los resultados obtenidos con el modelo y considerar los siguientes pasos a llevar a cabo para mejorar la detección de ataques.
- 4.– Investigar sobre otros IDS y NIDS para conocer el estado de esta tecnología en la actualidad.
- 5.– Realizar un análisis de los requisitos de la aplicación.
- 6.– Diseñar la aplicación, incluyendo la arquitectura empleada, diagramas de clases y tecnología a incluir.
- 7.– Desarrollar la aplicación. Este hito se divide en varios objetivos:
 - 7.1.– Visualizar el tráfico en la red.
 - 7.2.– Integrar el modelo obtenido para la detección de ataques.
 - 7.3.– Obtener estadísticas sobre los ataques detectados por el modelo.
 - 7.4.– Bloquear una dirección IP sospechosa.
 - 7.5.– Asegurar la independencia entre la aplicación y el modelo, con el objetivo de poder sustituir el modelo obtenido por otros más precisos en un futuro.
 - 7.6.– Buscar la mayor facilidad de despliegue e integración.
 - 7.7.– Garantizar la seguridad de la aplicación.
- 8.– Diseñar una guía de usuario.
- 9.– Realizar un análisis de objetivos futuros para la mejora de la aplicación.

1.3. Estructura del documento

A continuación se detalla el contenido de los distintos capítulos en que se divide este documento.

En el capítulo actual se incluye una breve introducción al proyecto, en la que se exponen las motivaciones y los objetivos del mismo.

En el capítulo 2 se presenta un análisis del estado actual de la tecnología empleada. Hablaremos de cómo funciona un Intrusion Detection System (IDS) y los distintos ejemplos de aplicaciones que han servido como inspiración. Además, justificaremos la aportación del proyecto en este área.

En el capítulo 3 se expone el contenido relacionado con la obtención del modelo de aprendizaje automático para la detección de ataques, presentando los *datasets* utilizados y la metodología empleada.

En el capítulo 4 se presentan los resultados obtenidos al analizar los *datasets* y generar modelos de detección.

En el capítulo 5 se expone el diseño de la aplicación. Hablaremos de los módulos en los que está dividida, el análisis de requisitos realizado, las clases de la aplicación y su arquitectura.

El capítulo 6 incluye el contenido relacionado con el desarrollo de la aplicación. Desde una justificación de la tecnología empleada, hasta una explicación en detalle de la metodología, la infraestructura y el funcionamiento de la aplicación.

Finalmente, en el capítulo 7 se presentan las conclusiones obtenidas, así como los objetivos futuros para mejorar la aplicación.

Además, se incluyen nueve apéndices que recogen contenido adicional del proyecto.

El apéndice A contiene una tabla con las características extraídas usando la herramienta CICFlowMeter para entrenar un modelo de aprendizaje automático.

El apéndice B incluye los requisitos funcionales de la aplicación.

El apéndice C contiene una descripción de las diferentes tecnologías empleadas en el desarrollo de VISION, junto con una breve justificación del motivo de su elección.

En el apéndice D se explica la estructura de ficheros de la aplicación.

El apéndice E contiene las listas de métodos para comunicarse con las APIs NIDS y Frontend, junto con una descripción de cada uno.

En el apéndice F se incluyen algunos ejemplos de adición de funcionalidades a la aplicación con el objetivo de ilustrar la escalabilidad del proyecto.

El apéndice G incluye una descripción de los diferentes criterios de seguridad seguidos durante el desarrollo de VISION.

El apéndice H contiene un manual de uso de la aplicación diseñado para guiar al usuario a la hora de instalar, desplegar y utilizar VISION.

Finalmente, el apéndice I incluye una explicación del uso de la herramienta *IPTables* y su funcionamiento.

1.4. Contenido del proyecto

El código del proyecto se puede encontrar en el siguiente repositorio de GitHub:

`https://github.com/Nagomez97/ML-NIDS`

En él se incluye un manual de instalación y uso de la aplicación, junto con el código necesario para entrenar un modelo y un *dataset* propio, llamado **VISION-IDS2020**, que es el que se ha empleado para obtener el modelo de detección definitivo integrado en la aplicación.

Se ha creado una página de documentación **GitHub Pages** con un manual de uso y de instalación.

`https://nagomez97.github.io/ML-NIDS/`

Además, se han preparado dos vídeos que muestran el funcionamiento de la aplicación. El primero de ellos incluye una muestra general de Vision.

`https://www.youtube.com/watch?v=4d_Rn7cw3kE`

El segundo vídeo muestra una situación en la que una máquina está realizando un ataque de fuerza bruta por FTP. VISION detecta el ataque y permite que el usuario bloquee la fuente del mismo.

`https://www.youtube.com/watch?v=u2BsUN6g0ok`

SISTEMAS DE DETECCIÓN DE INTRUSIONES

En este capítulo se pretende dar al lector una visión general de la situación actual de los distintos Sistemas de Detección de Intrusiones existentes.

2.1. Trabajo relacionado

Un Sistema de Detección de Intrusiones, o IDS, es una herramienta capaz de detectar un acceso no autorizado a una red o sistema. Este tipo de herramientas se puede catalogar según diversos criterios.

Si nos fijamos en el objetivo de la herramienta, podemos distinguir entre **Host Intrusion Detection System**, que analizan, entre otras cosas, los logs del sistema, y **Network Intrusion Detection System**, que analizan capturas de tráfico en redes. Existen soluciones híbridas que analizan tanto información interna del host como de la red.

Si tenemos en cuenta la metodología empleada, podemos distinguir entre los **IDS basados en firmas**, que comparan la información recogida con una base de datos, y los **IDS basados en anomalías**, que monitorizan la actividad del sistema en busca de comportamientos anómalos que puedan indicar la presencia de un atacante.

Finalmente, podemos fijarnos también en el tipo de respuesta, distinguiendo entre **IDS pasivos**, que únicamente alertan de un posible ataque, e **IDS reactivos**, capaces de responder de forma activa ante una amenaza.

VISION podría considerarse entonces como un NIDS pasivo que emplea un modelo de detección de ataques basado en técnicas de *Machine Learning*. Además, cuenta con una interfaz que facilita la reacción ante amenazas, y es fácilmente escalable hasta conseguir cierta reactividad por parte del sistema.

A continuación presentamos algunos de los IDS más utilizados en la actualidad.

2.1.1. Snort

Snort es un NIDS gratuito y de código abierto. Basa su funcionamiento en la creación de reglas utilizando un lenguaje propio, que permiten detectar una gran cantidad de ataques. Además, puede utilizarse para monitorizar una red o registrar paquetes. Se trata de un sistema pasivo, ya que cuando un paquete coincide con alguno de los patrones definidos por el usuario, se reporta para que un analista pueda investigarlo.

2.1.2. Hogzilla

Hogzilla IDS es otra solución de código abierto, más moderna que Snort. Se trata en esta ocasión de un IDS basado en anomalías, lo que le permite detectar ataques *zero-day* (ataques desconocidos hasta la fecha). De nuevo, es capaz de detectar una gran cantidad de ataques. Su motor de detección de anomalías cuenta con modelos entrados usando diversos algoritmos, como por ejemplo *k-Means*.

2.1.3. Suricata

Suricata es otro IDS basado en reglas predefinidas, con la diferencia de que incluye un Intrusion Prevention System (IPS) o, en castellano, Sistema de Prevención de Intrusiones, capaz de mitigar amenazas automáticamente.

2.1.4. SolarWinds

SolarWinds es un IDS privado, cuyas licencias parten de los \$5000.00. De nuevo, se trata de un sistema basado en reglas, que incluye más de 700 reglas predefinidas.

2.2. Ataques en redes

A lo largo de este documento vamos a mencionar algunos ataques bastante comunes hoy en día. En esta sección explicamos su funcionamiento y las herramientas empleadas.

2.2.1. Denegación de Servicio (DoS)

Un ataque de denegación de servicio, más conocido por sus siglas en inglés *DoS*, es un ataque que causa que un servicio o recurso se vuelva inaccesible para otros usuarios. Generalmente, esta clase de ataque se realiza sobre servicios web. En la actualidad, lo más común es encontrar ataques

de denegación de servicio distribuidos, llamados así porque se llevan a cabo desde varios nodos (generalmente una red de ordenadores secuestrados por *malware*, conocida como *botnet*).

En la mayoría de los casos, un ataque de este estilo se realiza saturando el servidor con peticiones para consumir un recurso del mismo.

La herramienta que hemos utilizado para realizar este tipo de ataque se llama **hulk** [3].

2.2.2. Escaneo de puertos

Aunque no se trata específicamente de un ataque, un escaneo de puertos suele ser siempre la primera fase de un test de penetración (*pentest*). Consiste en analizar los diversos puertos de un host empleando para ello múltiples técnicas, con el objetivo de obtener información relevante para un posterior ataque.

La herramienta más utilizada para este tipo de análisis es **nmap** [4].

2.2.3. Ataque por fuerza bruta

Uno de los ataques más comunes en la actualidad consiste en la obtención de credenciales utilizando métodos de fuerza bruta (esto es, probar todas las credenciales posibles hasta tener éxito). Sin embargo, lo más corriente es que se utilicen ataques por diccionario, que consisten en generar previamente un conjunto de millones de palabras (lo que se denomina diccionario o *wordlist*) y probarlas una a una. De este modo, se evita el usar palabras sin sentido, como `aaaaaaaaa`.

La herramienta que hemos utilizado para realizar este ataque se llama **THC-Hydra** [5].

2.3. Aportación del proyecto

Analizando las diversas soluciones existentes, nos dimos cuenta de que prácticamente no existen NIDS basados en firmas que, en vez de funcionar por reglas predefinidas, sean capaces de detectar ataques gracias a un modelo de detección entrenado usando técnicas de aprendizaje automático. Además, muchos de estos IDS están ideados para funcionar sobre la infraestructura de una gran compañía, habiendo muy pocas soluciones enfocadas a la seguridad del usuario estándar.

VISION pretende situarse como una solución de código abierto que permita a un usuario entrenar su propio modelo para funcionar de manera óptima en una red particular. Consideramos, por tanto, que la aplicación podría llegar a convertirse en una herramienta muy útil para la seguridad doméstica, especialmente a día de hoy, cuando empezamos a adentrarnos en la era del Internet of Things (IoT) y la ciberseguridad sigue siendo una de las principales tareas pendientes del hogar.

MODELO DE DETECCIÓN AUTOMÁTICA DE INTRUSIONES

En el presente capítulo se detalla la metodología empleada para obtener modelos de aprendizaje automático con el objetivo de resolver el problema actual: diseñar un sistema capaz de detectar ataques en redes.

Históricamente, los sistemas de detección de intrusiones se han basado en analizar un conjunto de ataques y obtener *signatures* o detalles que los caractericen para, seguidamente, diseñar una serie de filtros capaces de detectarlos en un futuro.

Aunque hoy en día la tendencia consiste en emplear algoritmos no supervisados, capaces no sólo de detectar ataques conocidos, sino de alertar de ataques nuevos, el objetivo de este proyecto es el de diseñar un sistema de aprendizaje supervisado, empleando para ello *datasets* diseñados con este propósito.

3.1. Dataset CSE-CIC-IDS2018

El primer *dataset* utilizado ha sido **CSE-CIC-IDS2018** [1], diseñado por el CIC en base al artículo de Sharafaldin, Lashkary y Ghorbani, “Towards Generating a New Intrusion Detection *dataset* and Intrusion Traffic Characterization” [6].

El CIC ya había presentado otros *datasets* similares en los años 2012 y 2017, para los que prepararon una infraestructura similar a la de una empresa, con varios departamentos, varias subredes y múltiples dispositivos. De ese modo, un grupo de usuarios generaba tráfico benigno de una forma variada, teóricamente abstracta y real durante días, obteniendo resultados que podrían ser extrapolables.

En esta tercera versión, además de incluir nuevos ataques, decidieron trabajar empleando modelos de aprendizaje automático que habían sido entrenados tanto para generar tráfico benigno, como ataques específicos. Por lo tanto, este conjunto de datos no ha sido generado por humanos, sino por sistemas entrenados.

Desde el enlace de Amazon Web Service (AWS) disponible en la página web [1] es posible des-

cargar tanto los *pcap* como los *CSV*. En nuestro caso, descargamos directamente los *CSV* puesto que los archivos tipo *pcap* ocupan más de 100 GB.

En la siguiente tabla se detallan los ataques que incluye el *dataset*, así como su duración y las herramientas utilizadas.

Attack	Tools	Duration	Attacker	Victim
Bruteforce attack	FTP – Patator	One day	Kali linux	Ubuntu 16.4 (Web Server)
	SSH – Patator			
DoS attack	Hulk, GoldenEye, Slowloris, Slowhttptest	One day	Kali linux	Ubuntu 16.4 (Apache)
DoS attack	Heartleech	One day	Kali linux	Ubuntu 12.04 (Open SSL)
Web attack	<ul style="list-style-type: none"> • Damn Vulnerable Web App (DVWA) • In-house selenium framework (XSS and Brute-force) 	Two days	Kali linux	Ubuntu 16.4 (Web Server)
Infiltration attack	<ul style="list-style-type: none"> • First level: Dropbox download in a windows machine • Second Level: Nmap and portscan 	Two days	Kali linux	Windows Vista and Macintosh
Botnet attack	<ul style="list-style-type: none"> • Ares (developed by Python): remote shell, file upload/download, capturing screenshots and key logging 	One day	Kali linux	Windows Vista, 7, 8.1, 10 (32-bit) and 10 (64-bit)
DDoS+PortScan	Low Orbit Ion Canon (LOIC) for UDP, TCP, or HTTP requests	Two days	Kali linux	Windows Vista, 7, 8.1, 10 (32-bit) and 10 (64-bit)

Figura 3.1: Tabla de ataques incluidos en el dataset.

El tráfico fue capturado usando un *sniffer* en formato *.pcap*. Sin embargo, para poder trabajar sobre el *dataset* es necesario extraer características numéricas del mismo, para lo que se usa **CICFlowmeter**.

3.1.1. CICFlowmeter

CICFlowmeter es una herramienta escrita en Java, diseñada por el CIC [7]. Permite obtener más de 80 valores estadísticos por cada entrada, denominada *flow*.

Un *flow* es una agrupación de paquetes de red que pertenecen a una misma comunicación. Un *flow* TCP, por ejemplo, engloba al conjunto de paquetes de una conexión TCP entre dos hosts desde que se inicia dicha conexión hasta que esta se cierra (o bien porque haya terminado, o bien por *timeout*). El final de un *flow* UDP siempre viene determinado por un *timeout*. En el apéndice A se incluye una tabla explicando las características extraídas y su significado.

Pese al buen funcionamiento de la herramienta, fueron necesarios algunos ajustes, como modificar el formato de fecha para poder ordenar *flows* en la base de datos, o solucionar algunos errores tipográficos. La versión definitiva corregida, empleada en la aplicación, se encuentra en el [repositorio de GitHub de VISION](#).

3.2. Preprocesamiento

Como ya se ha mencionado, en el proyecto de Colab se cargaron directamente los CSV desde la nube. Sin embargo, seguían ocupando más espacio en memoria del disponible (25 GB máximo). Por lo tanto, fue necesario reducir el tamaño de *dataset*.

En un primer intento, simplemente tomamos una muestra aleatoria del 20 % del *dataset* original. Sin embargo, de esta forma perdíamos la proporción inicial de cada tipo de ataque. Por lo tanto, decidimos tomar el 20 % del total de ataques de cada tipo, manteniendo esa proporción. A este *dataset* se le denomina `Sample1`. En la figura 3.2 se muestra la proporción entre los paquetes benignos y los ataques de la muestra.

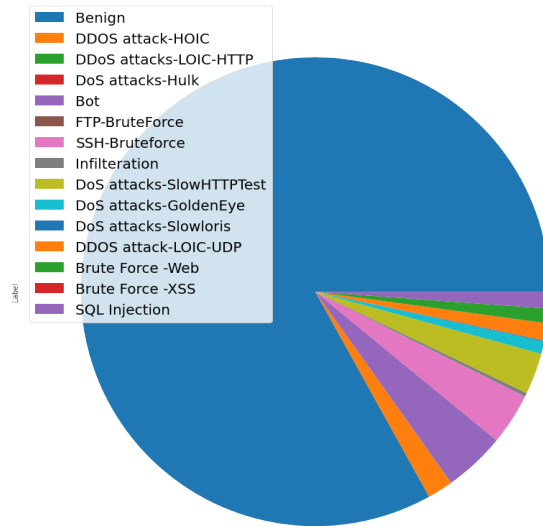


Figura 3.2: Proporción entre paquetes benignos y ataques en la muestra *Sample1*.

Como el objetivo es el de entrenar un modelo capaz de detectar ataques (nos da igual el tipo de ataque que sea), decidimos simplificar el *dataset* aplicando una etiqueta binaria: 1 para los ataques y 0 para los benignos.

Más adelante nos dimos cuenta de que probablemente obtendríamos mejores resultados con un *dataset* equilibrado, puesto que más del 75 % del tráfico capturado era benigno. Por lo tanto, se generaron las muestras *Sample2*, *Sample3* y *Sample4* tomando un 40 %, un 20 % y un 5 %, respectivamente, de cada tipo de ataque. A continuación, se sumó el número total de ataques recolectados y se tomó el mismo número de paquetes benignos. De esa forma conseguimos un *dataset* donde el 50 % de los paquetes estaban etiquetados como ataques, manteniendo la proporción entre cada tipo de ataque. Estas muestras tienen un tamaño de aproximadamente 8 %, 4 % y 1 % del *dataset* original, respectivamente.

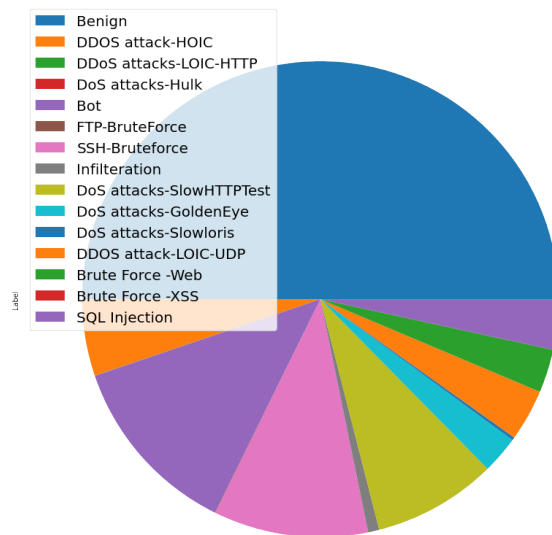


Figura 3.3: Proporción entre paquetes benignos y ataques en la muestra *Sample2*.

La figura 3.3 muestra la proporción entre ataques y benignos en la nueva muestra recortada y equilibrada.

Pese a la pequeña proporción de estos últimos *datasets*, seguimos teniendo del orden de 10^5 *flows* en cada uno, por lo que la muestra es bastante significativa.

El siguiente fragmento de código muestra cómo se ha recortado el *dataset*.

Código 3.1: En esta figura se presenta el código empleado para recortar el *dataset* original manteniendo la proporción de ataques y un equilibrio entre etiquetas.

```

1  for label in labels:
2      if label != 'Benign':
3          # This is new dataframe with only the 40% of the attacks, keeping the proportion between labels
4          attack_data = df.loc[df.Label == label].sample(frac=0.4)
5          day_dfs.append(attack_data)
6          total_attacks += attack_data.shape[0]
7      # We got all the attack dfs
8      # Now we want the same amount of benign flaws
9      total_benign = df.loc[df.Label == 'Benign'].shape[0]
10     attack_proportion = total_attacks / total_benign
11     benign_data = df.loc[df.Label == 'Benign'].sample(frac=attack_proportion)
12
13     day_dfs.append(benign_data)

```

También se eliminaron algunas entradas por diversos motivos. Las IPs y puertos de origen y destino aportarían información demasiado específica, que queremos evitar (un ataque no va a depender de la IP desde la que se realice dentro de una red). El *Timestamp* también es una característica a evitar. Por otro lado, había una etiqueta `Protocol` que en ningún momento se explicaba a qué correspondía ni cómo se había codificado, por lo que decidimos eliminarla.

3.3. Entrenamiento

Una vez se ha cargado el *dataset*, procedemos a entrenar un modelo capaz de distinguir entre las etiquetas 1 (ataque) y 0 (benigno). En esta sección vamos a hablar del algoritmo de Regresión Logística y a explicar la implementación realizada.

3.3.1. Regresión Logística

El algoritmo de regresión logística permite solucionar problemas de clasificación. En concreto, en su modalidad binaria este algoritmo se adapta perfectamente a nuestras necesidades: conocer si un *flow* es benigno o forma parte de un ataque.

El principal motivo de la elección de este algoritmo es su simplicidad. Además, tras probarlo con los *datasets* disponibles y ver que los resultados eran más que satisfactorios y que el tiempo de ejecución era relativamente corto, decidimos que era suficiente para cumplir con nuestro objetivo. Una posible

extensión de este trabajo sería probar con modelos más sofisticados, o incluso con un enfoque distinto usando modelos no supervisados.

La regresión logística es una adaptación del algoritmo de regresión lineal para problemas de clasificación. Se emplea como entrada de la función sigmoïdal una función lineal:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

La función de probabilidad del modelo queda entonces definida de la siguiente manera:

$$P(Y = 1|X = x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \dots + \beta_i x_i)}} = \frac{e^{\beta_0 + \beta_1 x_1 + \dots + \beta_i x_i}}{1 + e^{\beta_0 + \beta_1 x_1 + \dots + \beta_i x_i}} \quad (3.1)$$

donde $P(Y = 1|X = x)$ denota la probabilidad de que la variable aleatoria Y valga 1 sabiendo que el predictor X tiene el valor $x = (x_1, \dots, x_i)$ (conjunto de características de una observación).

Para entrenar el modelo, se procede a ajustar los parámetros β_j por el método de *Descenso por Gradiente*.

A la hora de predecir, la función de probabilidad va a devolver un número entre 0 y 1. En general, si la probabilidad es $P > 0.5$ se predice un valor de la variable aleatoria igual a 1, aunque este umbral se puede modificar para una mayor (o menor) sensibilidad.

3.3.2. Hiperparámetros

Un hiperparámetro es un valor, generalmente numérico, que sirve como propiedad arbitraria de un algoritmo de aprendizaje. Un hiperparámetro no puede inferirse a partir de los datos, sino que tiene que ser el analista quien seleccione su valor. Generalmente, esto se hace por experimentación, probando combinaciones de hiperparámetros hasta obtener los mejores resultados.

Una de las técnicas de selección de hiperparámetros más utilizada es la búsqueda en malla, o *grid search*, que realiza una búsqueda exhaustiva entre todas las combinaciones posibles de hiperparámetros usando un producto cartesiano.

En nuestro caso, hemos optado por realizar una **búsqueda en malla por validación cruzada** (*cross-validation*) para contrastar cada combinación de hiperparámetros. Este proceso consiste en dividir el *dataset* de entrenamiento en una serie de *hojas* y usar de forma rotatoria cada hoja como conjunto de validación, mientras se usan el resto de hojas como conjunto de entrenamiento. El ejemplo más común es la validación cruzada de cinco hojas (*five-fold cross-validation*), que consiste en tomar cinco subconjuntos $\{F_1, F_2, F_3, F_4, F_5\}$ y entrenar así cinco modelos por cada combinación de hiperparámetros, obteniendo cinco resultados por cada combinación, uno por cada subconjunto. Se toma como puntuación la media de esas cinco puntuaciones y, una vez se conoce la mejor combinación, se entrena un modelo final con el *dataset* completo y los mejores hiperparámetros.

El hiperparámetro sobre el que vamos a trabajar es el parámetro **C**, que se emplea para añadir cierta penalización para cada punto mal clasificado. Este parámetro multiplica al término de error, por lo que si **C** es demasiado grande, el error tendrá demasiado peso, sobreajustando el modelo. Si **C** es demasiado pequeño, se le dará muy poca importancia al error. Por lo tanto, es necesario ajustar este parámetro a un valor intermedio, usando para ello el conjunto de validación.

3.3.3. Entrenamiento y Test

Para poder evaluar un modelo y comprobar su eficacia, es necesario contar con un conjunto de datos que sirva como test. Hemos optado por una aproximación estándar, que es dividir el *dataset* en una proporción de 70 % para entrenamiento y 30 % para test. De este modo, una vez se ha entrenado el modelo, es posible obtener medidas de su eficacia en forma de precisión, *recall*, Curva ROC y demás, usando el conjunto de test.

3.3.4. Estandarización

Con el objetivo de optimizar el rendimiento de nuestro algoritmo, es necesario *estandarizar* el *dataset*. La estandarización consiste en reescalar cada uno de los valores del conjunto de datos hasta conseguir una distribución normal, de media $\mu = 0$ y desviación típica $\sigma = 1$. El proceso de estandarización es el siguiente

$$\hat{x}^{(j)} = \frac{x^{(j)} - \mu}{\sigma}, \quad (3.2)$$

donde μ es la media del vector x , σ su desviación típica, y $x^{(j)}$ representa el elemento j -ésimo del vector de características.

3.3.5. Implementación

El entrenamiento se ha realizado utilizando *sklearn* y el algoritmo de regresión logística.

La librería *sklearn* cuenta con una función de *Pipeline* que permite unir el proceso de estandarización con el algoritmo de regresión logística. De este modo, nos aseguramos de que los mismos parámetros empleados para estandarizar el conjunto de entrenamiento se usan también sobre los conjuntos de predicción (no tiene sentido estandarizar utilizando medias y desviaciones distintas, puesto que los datos no serían comparables). Además, el hecho de disponer de un único objeto capaz de preprocesar los datos y predecir en una misma llamada hace mucho más sencillo implantarlo en nuestra aplicación.

Como ya hemos mencionado, se realiza una búsqueda exhaustiva sobre el hiperparámetro **C** usan-

do la función *GridSearchCV*, con una validación cruzada en cinco hojas. En el siguiente fragmento se incluye el código utilizado para entrenar un modelo.

Código 3.2: Código empleado para entrenar un modelo con el algoritmo de Regresión Logística.

```

1  from sklearn.model_selection import train_test_split
2  from sklearn.model_selection import GridSearchCV
3  from sklearn.linear_model import LogisticRegression
4  from sklearn import metrics
5  import time
6  from sklearn.pipeline import Pipeline
7
8  param_grid = [
9      {
10         'logistic__C' : [0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000], # valores para C
11     }
12 ]
13
14 start_time = time.time()
15 print("Grid_Search...", end="")
16
17 # Obtenemos conjuntos de entrenamiento y test
18 Y = df['Label'].to_list()
19 X = df.drop(['Label'], axis=1)
20 X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3, stratify=Y)
21
22 # Creamos un pipe que estandarice los datos antes del entrenamiento
23 logreg = LogisticRegression(max_iter=20000)
24 pipe = Pipeline(steps=[('standardscaler', StandardScaler()), ('logistic', logreg)])
25
26 gridsearch = GridSearchCV(pipe, param_grid, cv=5, verbose=3)
27
28 gridsearch.fit(X_train, np.array(y_train))
29
30 elapsed_time = time.time() - start_time
31
32 gridsearch.data_header = df.columns
33
34 print("Finished._Elapsed_time:_{:d}_min_{:d}_seg".format(int(elapsed_time/60), int(elapsed_time
    % 60))

```

El modelo resultante se almacena mediante *pickling* para poder importarlo desde la aplicación.

3.4. Dataset VISION-IDS2020

En el capítulo 4 hablaremos en detalle de los resultados obtenidos con el *dataset* CSE-CIC-IDS2018. Por el momento, adelantamos que se observó cierta dependencia sobre los datos utilizados para entrenar el modelo, concluyendo que la infraestructura simulada en ese *dataset* no es tan general como aseguraban sus creadores, por lo que el modelo entrenado no se comportaba de forma correcta

al desplegarlo en nuestra red.

Tras generar un pequeño *dataset* de prueba «casero» y observar que los resultados eran bastante más prometedores, se decidió proceder a elaborar un *dataset* más completo con el que entrenar un modelo eficaz. A este *dataset* le hemos llamado **VISION-IDS2020**.

En esta sección detallaremos la infraestructura y metodología empleadas para su obtención.

3.4.1. Infraestructura

El objetivo era conseguir simular un entorno de red realista, pero tenía que ser posible ejecutar un *sniffer* desde el punto de acceso de la red, para poder monitorizar el tráfico completo. Además, era muy importante aislar la máquina víctima para poder etiquetar bien los ataques.

Se optó por utilizar una **Raspberry Pi** como punto de acceso, usando para ello la herramienta **RaspAP** [8], al que se conectaron diversos dispositivos con el objetivo de generar tráfico «real» y obtener así una muestra del comportamiento de la red. La Raspberry Pi estaba conectada al router por un cable RJ45 y desplegaba una subred para evitar interferencias. En la siguiente figura se muestra un esquema con la infraestructura utilizada.

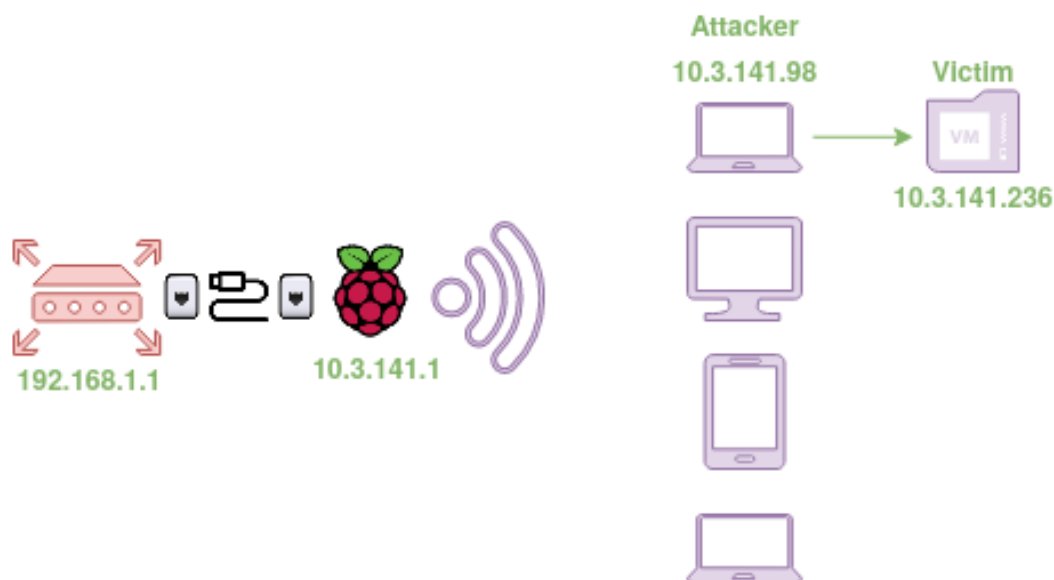


Figura 3.4: Esquema de la infraestructura empleada para la obtención del *dataset*.

Las direcciones IP relevantes son la **10.3.141.1**, correspondiente a la Raspberry Pi como punto de acceso, la **10.3.141.98**, correspondiente a la máquina atacante y la **10.3.141.236**, correspondiente a la víctima.

Con el objetivo de simular un dispositivo víctima realista, se desplegó una Máquina Virtual (VM) con una conexión de red puenteada, dentro de la propia máquina atacante. Además, se conectó una segunda tarjeta de red al host para evitar que las peticiones entre atacante y víctima se resolvieran de manera local.

Como víctima se usó un Ubuntu Server en el que se desplegó una sencilla web (puerto 80) con un Apache Server, un servidor File Transfer Protocol (FTP) (puerto 21) protegido con credenciales y un servidor Secure Shell (SSH) (puerto 22) accesible desde la red.

3.4.2. Metodología

Para recolectar información de una manera ordenada y fiable, se fijaron unas franjas horarias dentro de las cuales tendría lugar cada tipo de ataque. Durante el período de monitorización, todos los dispositivos (excepto la víctima) generan tráfico benigno que incluye visitas a páginas web, videollamadas, servicios de streaming, conexiones SSH y FTP... Los únicos paquetes benignos que puede recibir la VM son paquetes de configuración y descubrimiento, como por ejemplo Address Resolution Protocol (ARP), que, inevitablemente, quedarán catalogados como ataques. Sin embargo, la presencia de estos paquetes mal etiquetados es ínfima y, además, todos los demás dispositivos van a generar tráfico similar, por lo que esperamos que se compense de cara a nuestro modelo.

En el primer *dataset* de prueba, con una monitorización de tan solo 15 minutos, obtuvimos unos resultados muy buenos. Por lo tanto, consideramos suficiente una duración de una hora en la monitorización para conseguir un *dataset* que cumpla con nuestros objetivos.

Para capturar el tráfico desde la Raspberry Pi se utilizó la herramienta **tshark**.

Los ataques realizados fueron los siguientes:

- **Fuerza bruta sobre SSH.** Utilizando la herramienta Hydra con el diccionario *rockyou.txt*, se realiza un ataque de fuerza bruta para tratar de adivinar las credenciales de acceso al servicio SSH.

```
hydra -l victim -P /usr/share/wordlists/rockyou.txt 10.3.141.236
-t 20 ssh
```

- **Fuerza bruta sobre FTP.** De la misma manera, se trata de obtener las credenciales FTP.

```
hydra -l victim -P /usr/share/wordlists/rockyou.txt 10.3.141.236
-t 20 ftp
```

- **Escaneo de puertos.** En este caso se utiliza la herramienta Nmap para realizar un escaneo completo de puertos de la víctima.

```
nmap -A 10.3.141.236
```

- **Denegación de Servicio.** Este último ataque emplea la herramienta Hulk [3] para realizar una Denegación de Servicio (DoS) sobre el puerto 80 de la víctima.

```
go run hulk.go -site http://10.3.141.236
```

La siguiente tabla recoge las distintas franjas horarias y los ataques realizados en cada una.

Comienzo	Fin	Explicación
19:10	20:10	Captura de tráfico
19:15	19:17	Nmap
19:20	19:29	Fuerza Bruta SSH
19:30	19:39	Fuerza Bruta FTP
19:40	19:45	DoS Hulk
19:46	19:48	Nmap
19:50	19:59	Fuerza Bruta SSH
20:00	20:10	Fuerza Bruta FTP

Tabla 3.1: Franjas horarias de los ataques realizados para nuestro *dataset* .

3.4.3. Resultado

En un primer intento, la VM se desconectó tras realizar el ataque DoS, por lo que fue necesario repetir el proceso. Sin embargo, el *dataset* obtenido resultó bastante útil como conjunto de pruebas para los modelos posteriores.

El resultado del segundo intento fue un pcap de 426 MB que, tras pasarlo por el CICFlowMeter, se convirtió en un CSV de 9 MB. Esta reducción tan drástica se debe a que la mayoría del tráfico generado es TCP, con sesiones de larga duración, por lo que cada *flow* engloba una gran cantidad de tráfico.

El trabajar usando franjas horarias no sólo permite aislar el tráfico, sino que además hace posible etiquetar cada ataque por separado. Como se aprecia en la figura 3.5, la proporción entre ataques y benignos está bastante equilibrada, y todos los tipos de ataques tienen una representación significativa, por lo que podemos esperar buenos resultados usando este *dataset* .

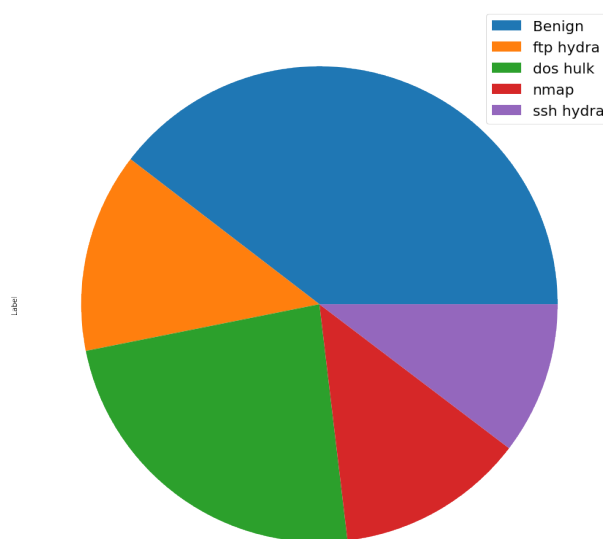


Figura 3.5: Figura que muestra la proporción entre tráfico benigno y cada tipo de ataque en el *dataset* generado.

RESULTADOS EXPERIMENTALES

En este capítulo se explican los resultados obtenidos al aplicar la metodología tratada en el capítulo anterior. Se van a contrastar los diversos modelos empleados, analizando además su comportamiento al desplegarlos sobre un entorno real.

4.1. CSE-CIC-IDS2018

En primer lugar vamos a analizar los resultados obtenidos al entrenar modelos con el *dataset* CSE-CIC-IDS2018, presentado en el apartado 3.1. En dicha sección ya mencionamos que, por motivos de tiempo y tamaño, fue necesario recortar el *dataset* original, generando tres muestras: *Sample2* (8 % del *dataset* original), *Sample3* (4 % del *dataset*) y *Sample4* (0.01 % del *dataset* original), todos ellos balanceados con el mismo número de ataques y benignos.

En el cuadro 4.1 se muestran los distintos modelos entrenados.

Nombre	Sample	% del original	GridSearch	Tiempo
LogReg1	Sample2	8 %	{1,10}	~ 1 h
LogReg2	Sample3	4 %	{0.01, 0.1, 1, 10, 100, 1000, 10000}	30 min
LogReg3	Sample4	1 %	{0.01, 0.1, 1, 10, 100, 1000, 10000}	15 min

Tabla 4.1: Modelos entrenados con el *dataset* CSE-CIC-IDS2018.

Los valores para el parámetro C son muy pocos (suelen utilizarse entorno a diez), pero como los resultados eran bastante buenos y el tiempo de ejecución aumentaba considerablemente con más parámetros, decidimos dejar una búsqueda más exhaustiva para un trabajo futuro.

Al compararlos con sus conjuntos de test, los tres modelos mostraron comportamientos similares. A continuación presentamos los datos del modelo LogReg2.

La matriz de confusión del cuadro 4.2 muestra unos ratios de verdaderos positivos/negativos muy buenos, fallando en apenas un 7 % de los casos.

	Positivos (%)	Negativos (%)
Verdaderos	48.05	45.41
Falsos	4.58	1.96

Tabla 4.2: Matriz de confusión del modelo LogReg2 entrenado sobre el CSE-CIC-IDS2018.

Se puede apreciar en el cuadro 4.7 una precisión del 96 % y un recall del 91 % en benignos, y una precisión del 91 % y recall del 96 % en ataques, lo que da a entender que el modelo es bastante bueno sobre el papel.

	Precision	Recall
Benignos	0.96	0.91
Ataques	0.91	0.96

Tabla 4.3: Report del modelo LogReg2 entrenado sobre el CSE-CIC-IDS2018. Refleja los resultados obtenidos sobre el conjunto de test.

El hecho de que el área bajo la curva ROC en la figura 4.1 sea de 0.93 confirma que el modelo se comporta adecuadamente sobre el conjunto de prueba.

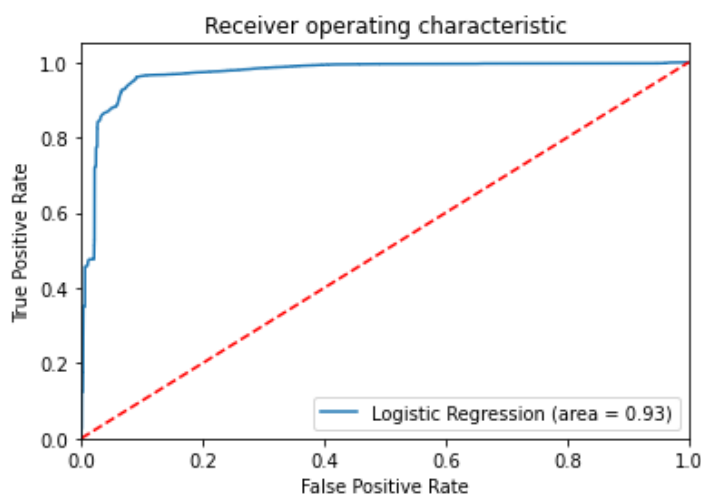


Figura 4.1: Curva ROC del modelo LogReg2 entrenado sobre el CSE-CIC-IDS2018.

Sin embargo, al desplegar el modelo sobre la aplicación y contrastarlo en un entorno real, nos encontramos con que su tasa de aciertos parece disminuir de forma drástica.

En un primer intento de generar el *dataset* VISION-IDS2020 nos encontramos con problemas de desconexión en algunas máquinas. Este intento tuvo lugar días antes del definitivo, con dispositivos diferentes conectados a la Raspberry Pi. Por lo tanto, decidimos usar los datos recogidos ese día como prueba de entorno real, y contrastar los modelos obtenidos contra este conjunto de datos.

Vemos en el cuadro 4.4 que el número de falsos positivos se ha incrementado en un 20 % con respecto al conjunto de test extraído del *dataset* CSE-CIC-IDS2018. Además, la curva ROC de la figura 4.2 también muestra unos resultados mucho peores en comparación.

	Positivos (%)	Negativos (%)
Verdaderos	47.34	27.29
Falsos	25.28	0.10

Tabla 4.4: Matriz de confusión al usar como conjunto de entrenamiento una muestra real.

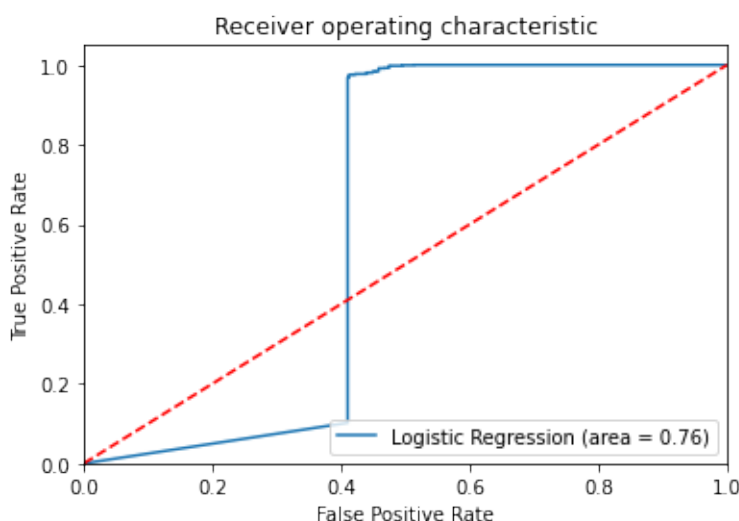


Figura 4.2: Curva ROC al usar como conjunto de entrenamiento una muestra real.

4.1.1. Conclusión

Aunque el modelo, a priori, parecía que iba a funcionar correctamente, al incluirlo en la aplicación y probarlo con tráfico real nos encontramos con todo lo contrario. La tasa de falsos positivos es tan elevada que hace imposible su implantación en el entorno de producción.

Para tratar de encontrar una explicación a esta diferencia entre resultados, analizamos los *datasets* en busca de diferencias sustanciales. La figura 4.3 muestra una gráfica que recoge las medias de cada característica en cuatro subconjuntos: ataques y benignos del *dataset* CSE-CIC-IDS2018 (en naranja y rojo, respectivamente), y ataques y benignos del *dataset* extraído de nuestra red (azul y verde, respectivamente).

Teniendo en cuenta que la gráfica está en escala logarítmica, las diferencias entre ambos *datasets* son considerables. Ciertos features presentan diferencias del orden de 10^8 . Por este motivo, concluimos que existe una fuerte dependencia sobre el *dataset* empleado, y que es muy posible que no podamos utilizar el CSE-CIC-IDS2018 para entrenar modelos que funcionen sobre otra infraestructura.

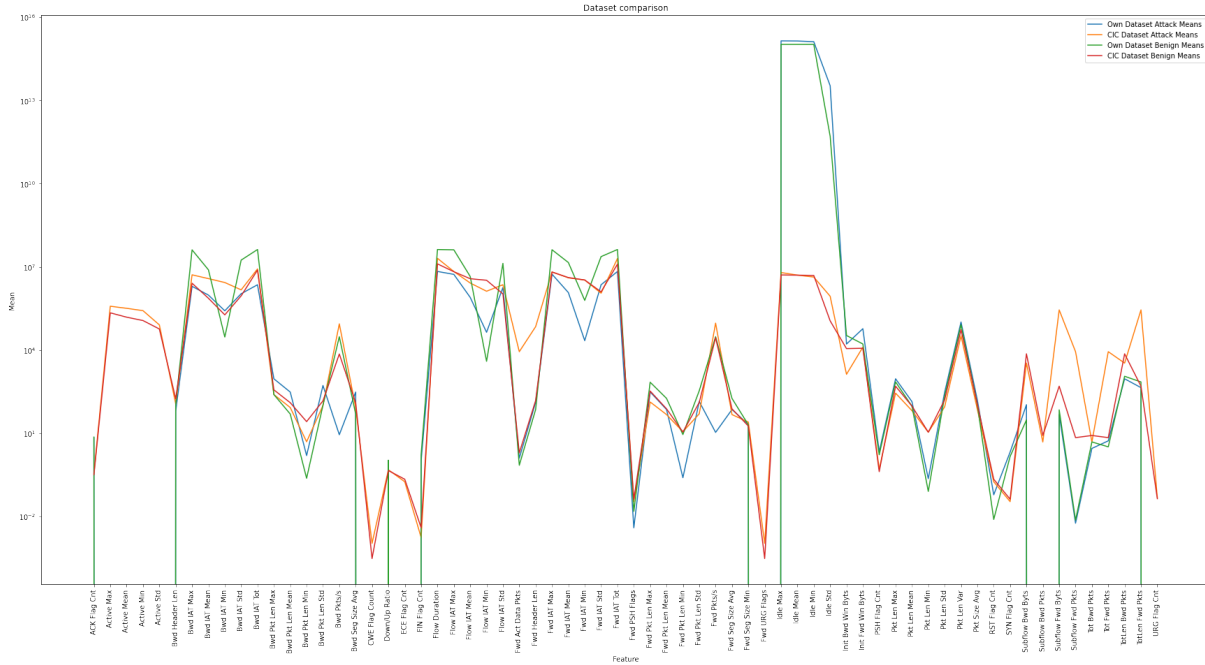


Figura 4.3: La gráfica recoge las medias de cada feature en el CSE-CIC-IDS2018 y el *dataset* propio.

Para descartar la posibilidad de que se estuviese produciendo un sobreaprendizaje sobre la muestra tomada del CSE-CIC-IDS2018, decidimos probar el modelo sobre otras muestras aleatorias del *dataset* completo. En todos los casos, los resultados fueron bastante buenos, similares a los que muestra la figura 4.4.

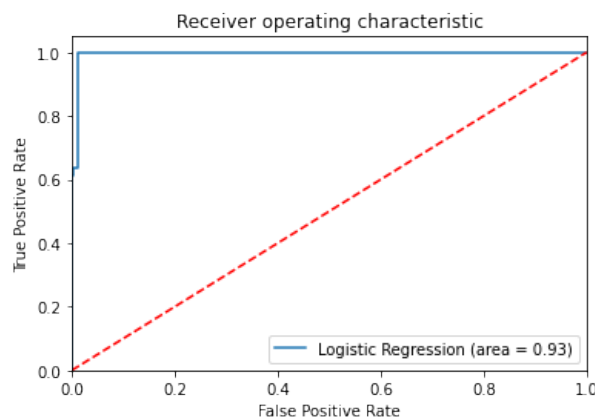


Figura 4.4: Resultado de probar el modelo LogReg2 sobre una muestra del día 14-02-2018 del *dataset* CSE-CIC-IDS2018.

Como el modelo entrenado desconocía estos datos (puesto que las muestras tomadas como prue-

ba corresponden a días completos que no estaban recogidos en la muestra de entrenamiento), pudimos descartar que se haya producido un sobreaprendizaje sobre el conjunto recortado de entrenamiento.

Todo esto nos llevó a pensar que, más que tratarse de un sobreajuste, el problema estaba en que la infraestructura de red y el tráfico recogidos en el *dataset* empleado no eran comparables al que estábamos generando desde nuestro laboratorio. En ese caso, si entrenábamos un modelo sobre nuestro propio *dataset*, obtendríamos unos resultados mucho mejores a largo plazo.

4.2. VISION-IDS2020

Como ya mencionamos en el apartado 3.4, a la vista de los resultados anteriores decidimos generar un *dataset* propio, denominado VISION-IDS2020, con la esperanza de demostrar que el problema de detección de ataques es dependiente de la infraestructura utilizada durante la obtención del conjunto de datos.

El *dataset* cuenta con 18 000 entradas etiquetadas como 1 (ataque) o 0 (benigno). En el cuadro 4.5 se puede ver el rango de valores para el hiperparámetro C escogidos.

Nombre	Sample	GridSearch	Tiempo
LogReg	VISION-IDS2020	{0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000}	5 min

Tabla 4.5: Modelos entrenados con el *dataset* VISION-IDS2020.

Los resultados obtenidos sobre el conjunto de test (30 % del *dataset*) demuestran un buen comportamiento del modelo sobre el papel.

De nuevo, la matriz de confusión del cuadro 4.6 muestra buenos ratios de verdaderos positivos/negativos, con pocos errores de clasificación.

	Positivos (%)	Negativos (%)
Verdaderos	45.12	48.65
Falsos	3.91	2.32

Tabla 4.6: Matriz de confusión del modelo entrenado sobre el VISION-IDS2020.

También apreciamos en el cuadro 4.7 una precisión del 95 % sobre benignos, 92 % sobre ataques, y un recall del 93 % sobre benignos, 95 % sobre ataques. En general, tenemos un ratio de aciertos del 94 %.

	Precision	Recall
Benignos	0.95	0.93
Ataques	0.92	0.95

Tabla 4.7: Report del modelo entrenado sobre el VISION-IDS2020.

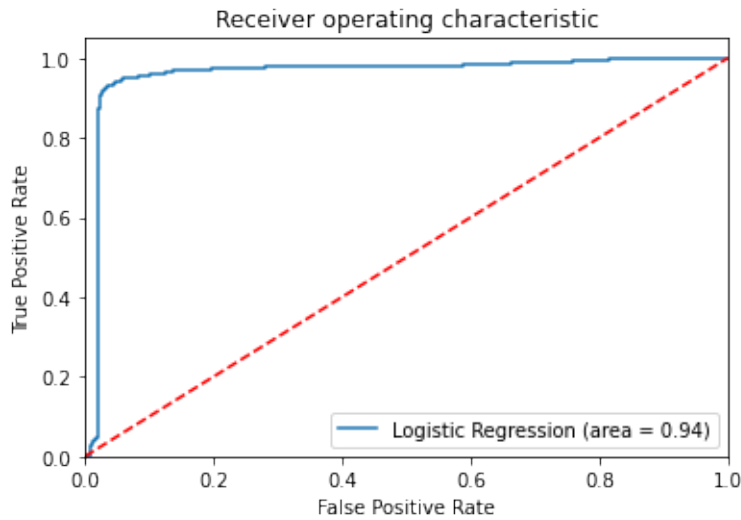


Figura 4.5: Curva ROC del modelo entrenado sobre el VISION-IDS2020.

El área bajo la curva ROC de la figura 4.5 es incluso mejor que en el caso del CSE-CIC-IDS2018, con un valor de 0.94.

Tenemos, por tanto, un modelo excelente teóricamente. Pero podría estar pasando lo mismo que en el caso anterior, por lo que resulta crucial probarlo en un entorno real antes de implantarlo en nuestra aplicación.

Para asegurarnos, usamos el conjunto de datos obtenido en el primer intento por generar un *dataset* definitivo, al que denominaremos **VISION-IDS2020-TEST**. De nuevo, destacamos que la obtención de este primer intento de *dataset* se produjo varios días antes de la obtención del *dataset* definitivo, realizando los ataques en horas distintas y durante períodos más cortos. Además, había dispositivos distintos conectados a la red, generando tráfico diverso. Por lo tanto, si el modelo se comporta de forma correcta sobre este conjunto de datos, podremos suponer que va a funcionar bien en general sobre la infraestructura propuesta.

La matriz de confusión del cuadro 4.8 presenta unos ratios de acierto bastante elevados, mientras que se siguen produciendo muy pocos errores.

	Positivos (%)	Negativos (%)
Verdaderos	49.03	3.54
Falsos	2.40	45.03

Tabla 4.8: Matriz de confusión del modelo entrenado sobre el VISION-IDS2020, sobre el conjunto VISION-IDS2020-TEST.

La curva ROC de la figura 4.6, de nuevo, presenta un área bajo la curva del 0.94, lo que es un buen indicador de la precisión del modelo.

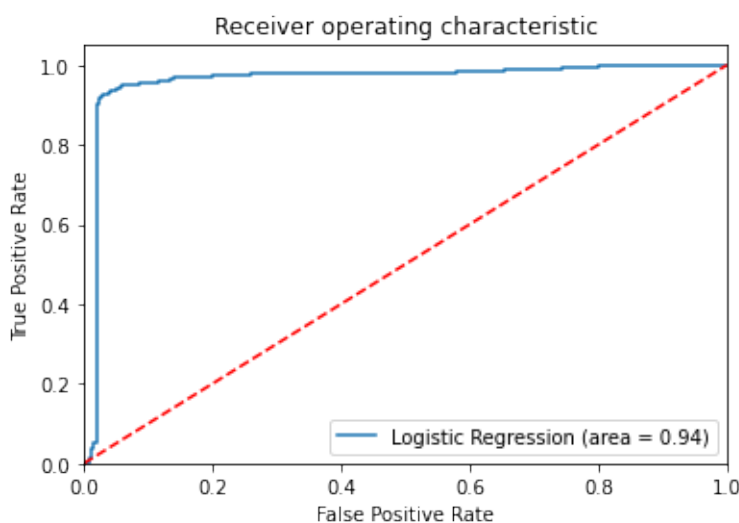


Figura 4.6: Curva ROC del modelo entrenado sobre el VISION-IDS2020, usando como conjunto de test el primer intento de obtención del dataset.

4.2.1. Conclusión

En general, parece que, al entrenar un modelo usando un *dataset* generado en la misma infraestructura en la que va a ser desplegado con posterioridad, obtendremos unos resultados mucho mejores.

De hecho, al implantar este modelo en la aplicación real, nos encontramos con que la cantidad de falsos positivos es ínfima mientras que, cuando realizamos uno de los ataques incluidos en el *dataset*, lo detecta prácticamente en todas las ocasiones.

En conclusión, podría decirse que nuestra aplicación necesita ser entrenada antes de desplegarse de forma definitiva. Además, viendo que utilizando el tráfico capturado durante tan solo una hora hemos obtenido unos resultados tan buenos, si el conjunto de datos de entrenamiento se trata con cuidado y se alarga en el tiempo durante algunos días, los resultados podrían ser más que satisfactorios.

Por último, queríamos estudiar el comportamiento de este nuevo modelo sobre el *dataset* CSE-

CIC-IDS2018. Tomamos para ello una muestra aleatoria del 20 % del *dataset*.

	Positivos (%)	Negativos (%)
Verdaderos	20.45	16.87
Falsos	33.13	29.55

Tabla 4.9: Matriz de confusión del modelo entrenado sobre el VISION-IDS2020, usando como conjunto de test el CSE-CIC-IDS2018.

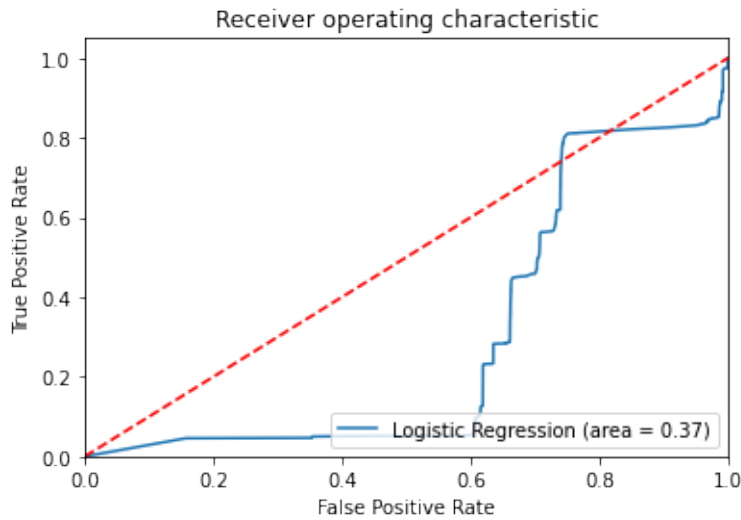


Figura 4.7: Curva ROC del modelo entrenado sobre el VISION-IDS2020, usando como conjunto de test el CSE-CIC-IDS2018.

Como era de esperar, los resultados mostrados en el cuadro 4.9 y la figura 4.7 son incluso peores que los obtenidos al contrastar el modelo anterior sobre una muestra real. De nuevo, apreciamos la fuerte dependencia del modelo con respecto a la infraestructura del *dataset*. Podemos confirmar entonces que nuestra teoría era correcta, y que será necesario entrenar un modelo en una infraestructura concreta antes de confiar en su precisión.

DISEÑO DE VISION

Desde un primer momento, el objetivo de este proyecto era no solo el de obtener un modelo capaz de detectar ataques en redes, sino el desarrollo de una aplicación completa que permita a un administrador mantener la seguridad en su red. A esta aplicación la hemos bautizado como **VISION**.

5.1. Módulos

La aplicación se ha dividido en varios módulos enfocados a una funcionalidad concreta, tal y como refleja la figura 5.1.

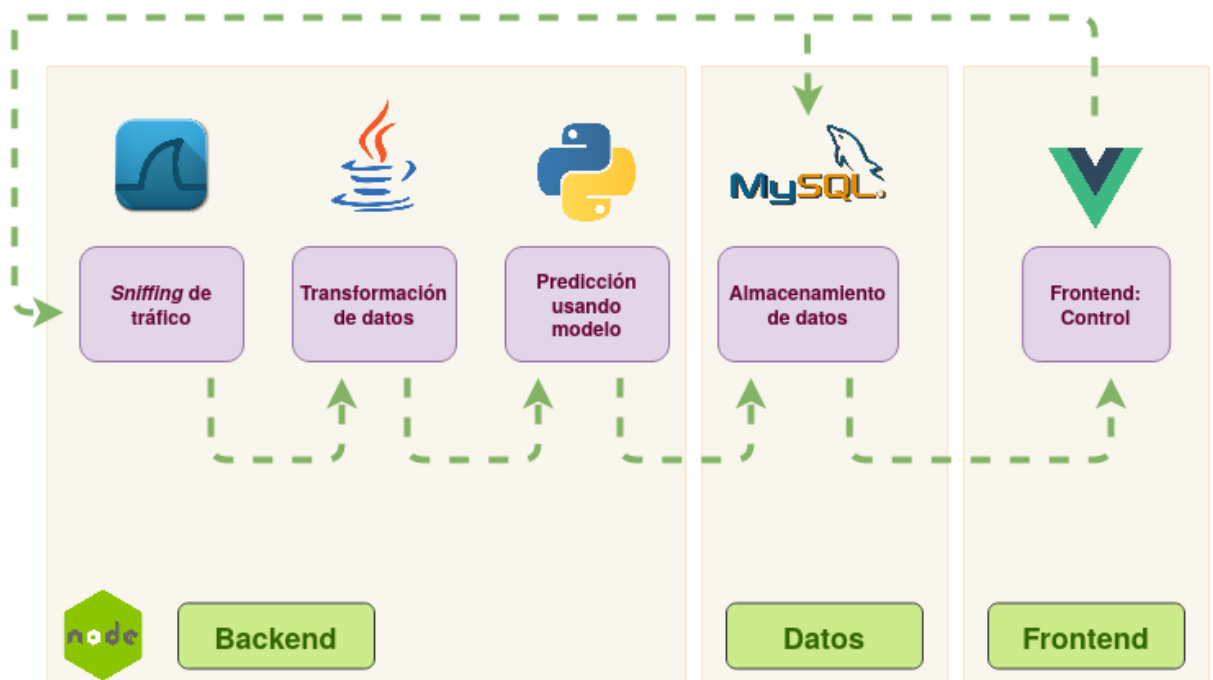


Figura 5.1: Diferentes módulos que componen la aplicación.

5.1.1. Módulo *Backend*

Este módulo, que se ejecuta en su totalidad desde el lado del servidor, incluye toda la funcionalidad de captura de tráfico, transformación de datos, predicción/detección de ataques usando un modelo entrenado, almacenamiento en base de datos y, en general, incluye la lógica interna de la aplicación.

Su funcionalidad se puede dividir en tres submódulos:

- Captura de tráfico y predicción de ataques.
- Interacción con la base de datos.
- Lógica interna.

Submódulo de Captura y Predicción de Ataques

Encargado de capturar el tráfico en la red para después transformarlo en un CSV usando CIC-FlowMeter. Esto enlaza directamente con la funcionalidad de predicción de ataques, implementada en Python, que utiliza el modelo obtenido tal y como se ha presentado en los capítulos anteriores.

El resultado de la predicción es un *dataframe* que incluye información por cada *flow* etiquetado. Esta información se inserta en base de datos empleando el siguiente submódulo.

Submódulo de Interacción con la Base de Datos

Debido a la diversidad de peticiones que se hacen contra la base de datos, se ha decidido separar su funcionalidad en un módulo capaz de gestionarlas.

Lógica Interna

El último submódulo es un poco más general. Puesto que el sistema se ha diseñado siguiendo un esquema de API REST, es necesario contar con un módulo que gestione esta interfaz a nivel de servidor, sirviendo como enlace entre todos los demás módulos y submódulos.

5.1.2. Módulo *Frontend*

VISION se pensó, desde un primer momento, como una aplicación que permitiera a un usuario gestionar el tráfico en una red en base a las alertas lanzadas por el modelo de predicción de ataques. Es por eso que la presencia de una interfaz de control resulta esencial.

Este módulo recoge toda la funcionalidad relacionada con esa interfaz: gestión de sesiones de usuario, visualización de datos, interacción directa con el usuario...

5.2. Requisitos de la aplicación

Cuando se diseña una aplicación, es muy importante tener clara su funcionalidad. A la hora de hablar de requisitos, se suele distinguir entre dos clases: funcionales y no funcionales.

Los requisitos funcionales son aquellos que, en general, están relacionados con la entrada y salida de datos y una interacción directa con el usuario. Hacen referencia a la funcionalidad de la aplicación en un sentido estricto.

Los requisitos no funcionales, por su parte, son también conocidos como atributos de calidad. Especifican criterios que permiten juzgar el comportamiento de un sistema, como la seguridad, el rendimiento, el tiempo de respuesta...

En esta sección se detallan tanto los requisitos funcionales como los no funcionales.

5.2.1. Requisitos funcionales

Los requisitos funcionales se especifican utilizando un modelo de casos de uso, en el que detallamos el escenario principal de cada uno.

Sin embargo, debido a su extensión, en este apartado únicamente incluimos el nombre e identificador de cada uno de ellos. En el apéndice B se encuentran todos en detalle.

- RF-01** Creación de un nuevo usuario.
- RF-02** Inicio de sesión.
- RF-03** Iniciar/Pausar captura de tráfico.
- RF-04** Filtrar datos en la tabla de *flows*.
- RF-05** Acceder a estadísticas del tráfico.
- RF-06** Fijar una IP como objetivo.
- RF-07** Bloquear una IP.
- RF-08** Desbloquear una IP.
- RF-09** Cierre de sesión.

5.2.2. Requisitos no funcionales

La figura 5.2 recoge todos los requisitos no funcionales junto con una breve descripción.

RNF-01: Tráfico cifrado SSL	
Descripción	El servidor contará con un certificado SSL que permita cifrar las comunicaciones tanto con la interfaz web como entre endpoints de la API.
RNF-02: API autenticada por tokens	
Descripción	Únicamente se podrá consumir un servicio de la API si el solicitante cuenta con un token de autenticación. Este token se generará de forma aleatoria en cada inicio de sesión, almacenándose en la BBDD para poder así contrastarlo
RNF-03: Gestión segura de contraseñas	
Descripción	Al crear un nuevo usuario, la contraseña no se almacenará en texto plano. En su lugar, se generará un hash RSA-SHA256 con salt. Cada vez que se reciba una solicitud de inicio de sesión, se comprobará que el hash de la contraseña del solicitante coincide con el hash almacenado.
RNF-04: Reporte de errores producidos en el servidor	
Descripción	Si se produce algún error en el servidor cuyo resultado pueda afectar a la experiencia del usuario (error al cargar un flow, predecir ataques o similar), se mostrará un mensaje alertando al usuario.
RNF-05: Log general	
Descripción	Al desplegar la aplicación, la consola desde la que se haya desplegado el servicio mostrará logs con información a varios niveles: debug (relacionado con proceso de debug en la aplicación), info (información relevante, pero no indica ningún fallo o error), error (se ha producido un error).

Figura 5.2: Tabla de requisitos no funcionales y su descripción.

5.3. Clases

Aunque la infraestructura de la aplicación es relativamente compleja, el sistema de clases es bastante simple.

Por un lado, contamos con una clase **Usuario** cuya única funcionalidad es la de garantizar el uso de sesiones en la aplicación, puesto que no existe ningún tipo de información asociada a un usuario.

También existe una clase **Flow** que representa los *flows* generados por nuestro modelo para la detección de ataques. Un Flow, por lo tanto, cuenta con las siguientes propiedades:

ip_src (STRING) Dirección IP de origen del Flow.

ip_dst (STRING) Dirección IP de destino del Flow.

port_dst (INT) Puerto de destino del Flow.

label (STRING) Etiqueta resultado de la predicción: puede valer *Attack* o *Benign*.

prob (FLOAT) Resultado numérico de la predicción, entre 0 y 1. El modelo cuenta con una variable umbral, de modo que si *prob* es mayor que ese umbral, el Flow se etiqueta como *Attack*.

len_fwd (INT) Longitud del *flow* en sentido saliente en bytes.

len_bwd (INT) Longitud del *flow* en sentido entrante en bytes.

Finalmente, la clase **Target** se emplea para representar las IPs que han sido fijadas como objetivo. Cuenta con un parámetro *blocked*, booleano, que permite saber si el objetivo ha sido bloqueado o no en la red.

El diagrama de clases de la figura 5.3 incluye las relaciones entre todas las clases empleadas, así como sus atributos.

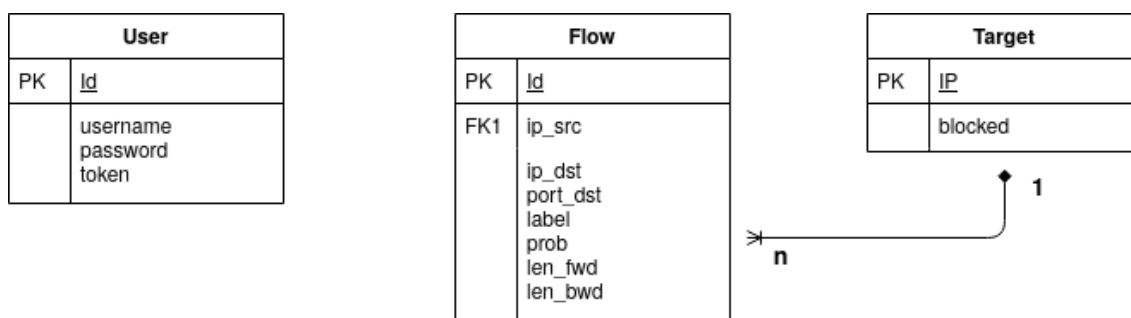


Figura 5.3: Diagrama de clases de la aplicación.

5.4. Arquitectura

Teniendo en cuenta los módulos presentados, así como las clases definidas, procedemos a explicar la arquitectura de la aplicación.

El modo en que se reparten los módulos deja patente que se ha seguido un modelo de desarrollo por capas a tres niveles:

Capa de presentación También denominada «capa de usuario» o «frontend», es la interfaz con la que interactúa directamente el usuario.

Capa de negocio En nuestro caso, es la capa principal, también conocida como «backend». Es la encargada de gestionar todas las peticiones que realiza el usuario desde la interfaz, y actúa como puente entre la capa de presentación y la de datos.

Capa de datos Es donde residen los datos de la aplicación y la lógica que permite el acceso a la base de datos.

DESARROLLO E IMPLEMENTACIÓN DE VISION

En este capítulo detallamos el proceso de desarrollo del software, teniendo en cuenta el diseño especificado en el capítulo 5. En primer lugar, se explica la infraestructura de la aplicación y la estructura de ficheros. También se profundizará en el desarrollo de cada uno de los tres niveles de la arquitectura por capas.

En el apéndice C se incluye una explicación de las tecnologías empleadas, justificando el uso de cada una.

La figura figura 6.1 representa las diferentes capas de la aplicación, junto con la tecnología empleada en cada una de ellas.



Figura 6.1: Arquitectura en tres capas de la aplicación

6.1. Infraestructura

Por cómo está diseñada VISION, pensamos que el mejor enfoque para su infraestructura es el de una **arquitectura orientada a microservicios**. Los servicios pueden estar escritos en lenguajes diferentes, e incluso ejecutarse sobre distintos sistemas operativos. Cada uno de ellos se encarga de

una funcionalidad concreta de la capa del negocio.

Decidimos, además, emplear contenedores Docker para desplegar los distintos microservicios de una manera sencilla y modular, tal y como se muestra en la figura 6.2.

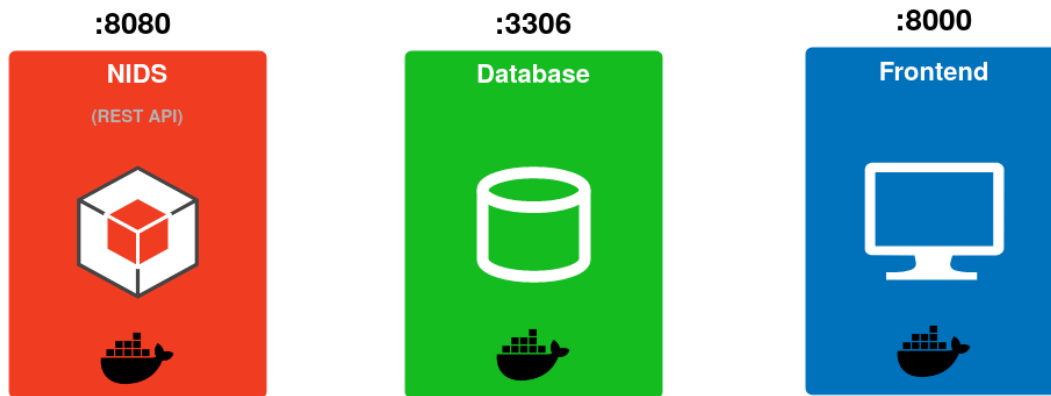


Figura 6.2: Diagrama que muestra cómo se reparten los servicios entre los diversos contenedores desplegados.

Para el despliegue se utiliza la herramienta *docker-compose*, que permite gestionar varios contenedores al mismo tiempo. Para ello se genera un fichero *docker-compose.yml* que incluye información sobre cada contenedor: nombre, variables de entorno, puertos, volúmenes enlazados...

Una observación importante es que, por defecto, el host de Docker virtualiza una red interna en la que despliegan los contenedores. En general, esta red está aislada del exterior. Sin embargo, el contenedor NIDS necesita tener acceso directo a la interfaz de red del host para poder capturar tráfico en la red externa. Para ello existe un modo de configuración de red, *network-mode: host*, que permite enlazar directamente el contenedor a un puerto del host, dándole acceso a la interfaz.

Como, además, necesitábamos que el contenedor Frontend fuese accesible desde el exterior para poder servir la página web, redirigimos el tráfico entre un puerto del host y uno del contenedor (aunque este sí se desplegó en la red interna de Docker).

Nos encontramos un pequeño problema al conectar el contenedor NIDS a la base de datos. Como NIDS ya no pertenece a la red interna de Docker, es necesario enlazar también la base de datos a un puerto del host, haciéndola accesible desde el exterior. Aunque lo ideal sería que únicamente fuese accesible desde el propio servidor, mientras las credenciales estén protegidas no consideramos el problema como algo crítico.

A continuación se describen los contenedores empleados en detalle.

6.1.1. NIDS

Este es el contenedor principal. Ejecuta una imagen de *Ubuntu Bionic* y contiene la capa de negocio de la aplicación. La red de este contenedor está enlazada a la del host (no vive dentro de la subred de Docker), puesto que el *sniffer* necesita tener acceso directo a la tarjeta de red del host.

Al desplegar el contenedor se instalan automáticamente todas las herramientas necesarias para su funcionamiento: un entorno de Java para el CICFlowMeter, Python y todas sus librerías que utilizamos y todos los paquetes de Node necesarios. Para instalar estos últimos se emplea el gestor de paquetes NPM (Node Packet Manager), que recoge todas las dependencias necesarias en un archivo *package.json*.

El contenedor NIDS engloba, por tanto, la lógica de negocio de la API, la librería de interacción con el ORM Sequelize, el módulo de predicción de ataques usando Python y el modelo entrenado, el módulo de captura de tráfico y el módulo de transformación de datos utilizando CICFlowMeter. Se encuentra enlazado al **puerto 8080** del host y cuenta con un certificado SSL para cifrar todas las comunicaciones con la API.

6.1.2. Database

Se trata del contenedor más ligero. Ejecuta una imagen MySQL que recibe las credenciales como variables de entorno incluidas en el fichero *docker-compose*. Uno de los pasos a seguir en el desarrollo de la aplicación pasaría por proteger estas credenciales, aunque no son accesibles por nadie ajeno a la máquina en la que se despliegan los contenedores. Está enlazado al **puerto 3306** del host.

6.1.3. Frontend

Este contenedor ejecuta una imagen de Node, muy ligera, y cuenta con un servidor web encargado de gestionar la interfaz de usuario.

Al separar el servidor de la interfaz del servidor NIDS conseguimos aligerar parte de la carga de trabajo de este último, puesto que cuando un usuario solicita recursos de la web (algo prácticamente constante), estos se sirven directamente desde el contenedor Frontend. Cuando se requiera información de la base de datos, por ejemplo, la conexión se realizará directamente con el servidor NIDS, enviando una petición protegida con un token.

El contenedor Frontend está enlazado al **puerto 8000** del host y también cuenta con un certificado SSL para cifrar todas las comunicaciones.

6.2. Estructura de ficheros

La estructura de ficheros se encuentra detallada en el apéndice D.

6.3. API

Realmente existen dos APIs: la principal, en el servidor NIDS y la del Frontend. En el apéndice E se detallan los métodos de cada una de ellas, junto con una breve descripción.

6.4. Escalabilidad

Pensamos que la aplicación puede tener un gran potencial, por lo que dotarla de una buena escalabilidad es uno de nuestros objetivos prioritarios. En el apéndice F explicamos algunos ejemplos sobre cómo incluir nuevas funciones a la aplicación.

6.5. Seguridad

VISION tiene acceso a recursos críticos de la red: desde la posibilidad de capturar tráfico de todo tipo hasta la capacidad de bloquear direcciones IP. Por lo tanto, gestionar correctamente la seguridad de la aplicación es un requisito indispensable.

En el apéndice G se detallan los criterios de seguridad seguidos durante el desarrollo de la aplicación. Se trata, quizás, del apéndice con el contenido más interesante, puesto que muchos de los criterios de seguridad allí descritos han supuesto modificaciones estructurales relevantes en el proyecto. Las diferentes secciones que lo componen son:

Seguridad en Docker Criterios de seguridad relacionados con el uso de contenedores.

Tráfico cifrado Justificación del empleo de certificados SSL para la aplicación.

Usuario único Justificación de la decisión de limitar el número de usuarios de la aplicación.

Seguridad en la API Probablemente el apartado más relevante. Explicación del funcionamiento de los *tokens* de autenticación en la aplicación.

Gestión segura de contraseñas Explicación del método de cifrado de contraseñas en base de datos.

Bloqueo de IPs Funcionamiento del sistema de bloqueo de IPs.

CONCLUSIONES Y TRABAJO FUTURO

En este capítulo se exponen las conclusiones obtenidas durante el desarrollo del proyecto, así como los posibles objetivos para mejorar la aplicación en un futuro.

7.1. Conclusiones

Se ha conseguido desarrollar una aplicación capaz de monitorizar el tráfico de una red, de predecir ataques con una precisión incluso mejor que la esperada en un principio y de bloquear tráfico malicioso. Los resultados son, por lo tanto, bastante satisfactorios, y cumplen con los objetivos marcados inicialmente.

Sin embargo, como se explica en el capítulo 4, llegamos a la conclusión de que el *dataset* CSE-CIC-IDS2018 no es una buena opción para entrenar modelos que vayan a ser desplegados en redes domésticas. Por lo tanto, se generó un *dataset* propio, llamado VISION-IDS2020, con el que se entrenó un modelo bastante preciso capaz de detectar ataques por fuerza bruta sobre SSH y FTP y ataques de denegación de servicio.

VISION se ha probado sobre un entorno real, generando tráfico benigno mientras se realizaban ataques puntuales, y en un alto porcentaje de los casos ha sido capaz de identificar las IPs emisoras de esos ataques, permitiendo que el usuario las bloqueara inmediatamente. Además ha sido sometida a diversas pruebas de integración y de uso, así como a una pequeña auditoría de seguridad, que han demostrado la robustez del sistema.

Una de las principales ventajas de la aplicación es que se ha diseñado con la escalabilidad en mente, sobretudo a la hora de incorporar nuevos modelos de detección de ataques. Por eso, tal y como especificamos en el apartado 6.4, resulta muy sencillo añadir nuevas funcionalidades o integrar un nuevo modelo.

7.2. Trabajo Futuro

Pese a que la aplicación cumple con todos los objetivos iniciales, no deja de ser una primera versión que, bien llevada, podría evolucionar hasta convertirse en un proyecto muy ambicioso. A lo largo del documento hemos mencionado ciertas carencias o posibles mejoras de la aplicación, que vamos a detallar a continuación.

En primer lugar, nos hemos centrado en entrenar únicamente modelos de Regresión Logística. Sería interesante probar otros algoritmos más complejos y comparar los resultados con los aquí obtenidos. Además, hemos enfocado el problema directamente usando algoritmos supervisados, cuando quizás un enfoque no supervisado o empleando técnicas de refuerzo podría convertir VISION en un IDS capaz de detectar ataques *zero-day* o desconocidos.

Además, el *dataset* VISION-IDS2020 cuenta con una cantidad limitada de ataques, por lo que sería necesario generar un *dataset* más completo para obtener unos mejores resultados.

Por otro lado, el rendimiento de la aplicación no es todavía todo lo bueno que cabría desear. Actualmente, el sistema procesa los *flows* de la última hora. Si en ese marco de tiempo se generase una gran cantidad de tráfico, al utilizar gráficas reactivas estaríamos sobrecargando la interfaz web, por lo que la navegación sería muy lenta y accidentada.

Otro punto a tener en cuenta es que VISION no se comporta como la mayoría de los IDS. No cuenta con un sistema eficaz de análisis de evidencias pasadas, sino que se centra en detectar ataques al momento. Además, al transformar las trazas de red a *flows*, perdemos toda información que pueda resultar relevante a la hora de realizar un análisis de incidencias. No tenemos forma, más allá de lo que detecta la propia aplicación, de analizar el contenido de los paquetes capturados en busca de evidencias.

Por lo tanto, un buen complemento para esta aplicación sería una herramienta de monitorización de tráfico capaz de almacenar los paquetes que han levantado alarmas en VISION para su posterior análisis.

Además, en el apéndice I hablamos de cómo VISION bloquea IPs usando *IPtables*, y de las limitaciones que esto conlleva. Una posible mejora a corto plazo podría ser el implementar una solución que permita bloquear tráfico a nivel de la capa de enlace, permitiendo que VISION sea capaz de mitigar ataques que se produzcan desde dentro de la red.

En conclusión, pretendemos que VISION evolucione con el paso del tiempo, recibiendo actualizaciones hasta convertirse en una opción relevante a la hora de proteger nuestra red doméstica.

BIBLIOGRAFÍA

- [1] Communications Security Establishment (CSE) and Canadian Institute for Cybersecurity (CIC), "CSE-CIC-IDS2018 on AWS." <https://www.unb.ca/cic/datasets/ids-2018.html>, 2018.
- [2] T. Mirza, "Building an intrusion detection system using deep learning." <https://towardsdatascience.com/building-an-intrusion-detection-system-using-deep-learning-b9488332b321>.
- [3] Grafov, "Hulk DoS Tool." <https://github.com/grafov/hulk>.
- [4] "Nmap: From beginner to advanced." <https://resources.infosecinstitute.com/nmap/>.
- [5] "Kali tools: Hydra package description." <https://tools.kali.org/password-attacks/hydra>.
- [6] Sharafaldin, I., Lashkari, A. H., and Ghorbani A. A., *Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization*. 4th International Conference on Information Systems Security and Privacy (ICISSP), 2018.
- [7] Canadian Institute for Cybersecurity (CIC), "CICFlowMeter." <https://github.com/CanadianInstituteForCybersecurity/CICFlowMeter>.
- [8] Billz, "RaspAP WiFi Configuration Portal." <https://github.com/billz/raspap-webgui>.
- [9] Google, "V8 Open Source High-Performance Engine." <https://v8.dev/>.
- [10] A. Bettini, "Vulnerability exploitation in docker container environments." Presented at Black Hat Europe, 2015.
- [11] Keycloak, "Keycloak: Open Source Identity and Access Management." <https://www.keycloak.org/>.
- [12] OWASP, "Password Storage Cheat Sheet." https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html.

DEFINICIONES

característica Dado un conjunto de datos, se conoce como característica a cada uno de los valores de una observación.

colab Herramienta de Google similar a un Jupyter Notebook que permite ejecutar código Python en la nube, con mayor potencia y capacidad de cálculo. Muy usado para aprendizaje automático.

conexión de red puenteada Modo de conexión a la red en una máquina virtual en el que la máquina recibe una dirección IP dentro de la red a la que esté conectado el host. Esto permite una conexión directa entre cualquier dispositivo de esa red y la máquina virtual, sin la existencia de un NAT de por medio. Cabe destacar que esta conexión no se hace de forma física (la máquina y el host, al fin y al cabo, comparten la misma tarjeta de red).

CSV Archivos de texto que incluyen separadores (comas, puntos y coma, tabuladores...) empleado para representar datos en forma de tabla.

Curva ROC Método para medir el rendimiento de un modelo de aprendizaje. Usa una combinación del recall y el ratio de falsos positivos para generar una imagen en forma de curva. Cuanto mayor sea el área bajo la curva (AUC), mejor será el clasificador.

dataframe Estructura de datos en forma matricial plana o de tabla empleada para el tratamiento directo de datos por librerías como Pandas.

endpoint Cada uno de los extremos de una comunicación.

flow Resultado de agrupar una serie de paquetes de red pertenecientes a una misma comunicación.

hash Una función *hash* es un algoritmo que permite transformar de forma unidireccional una cadena de texto en una cadena de longitud fija que no tiene relación aparente con el texto original.

hiperparámetro Valor, generalmente numérico, que sirve como propiedad arbitraria de un algoritmo de aprendizaje. Un hiperparámetro no puede inferirse a partir de los datos, sino que tiene que ser el analista quien seleccione su valor.

Hulk Herramienta para realizar ataques de denegación de servicio sobre páginas web o servicios HTTP.

Hydra Herramienta para realizar ataques de fuerza bruta.

IP Dirección de un dispositivo dentro de una red, en formato X.X.X.X.

IPTables Utilidad de línea de comandos de Linux que permite configurar el cortafuegos del sistema.

Nmap Herramienta que permite escanear redes y puertos, así como los servicios que se están ejecutando en cada máquina.

OWASP Proyecto de código abierto dedicado a investigar vulnerabilidades en software y combatir sus causas. Cuenta con una gran cantidad de manuales y documentos para ayudar a los desarrolladores a crear código seguro.

pcap Archivos binarios con extensión *.pcap* que contienen tráfico de red.

pickling Se conoce por '*Pickling*' al proceso de serialización de objetos en Python.

precisión Valor numérico para medir la eficacia de un modelo entrenado. Responde a la pregunta: *¿Cuál es el porcentaje de positivos acertados?*

Raspberry Pi Dispositivo consistente en una única placa que cuenta con CPU, RAM y todo lo necesario para actuar como un ordenador de pequeño tamaño y coste mínimo.

recall Valor numérico para medir el ratio de predicciones positivas correctas sobre el total de positivos. Responde a la pregunta: *De todos los positivos reales, ¿cuántos has acertado?*

sklearn Biblioteca para aprendizaje automático disponible en Python.

sniffer Software empleado para capturar tráfico en una red (*sniff*). El ejemplo más conocido es *Wireshark*.

tshark Herramienta de captura de tráfico en red.

ACRÓNIMOS

ARP	Address Resolution Protocol
AWS	Amazon Web Service
CIC	Canadian Institute for Cybersecurity
DoS	Denegación de Servicio
FTP	File Transfer Protocol
HIDS	Host Intrusion Detection System
IDS	Intrusion Detection System
IoT	Internet of Things
IPS	Intrusion Prevention System
MITM	Man-In-The-Middle
NIDS	Network Intrusion Detection System
NPM	Node Packet Manager
ORM	Objetc-Relational Mapping
SSH	Secure Shell
SSL	Secure Sockets Layer
VM	Máquina Virtual

APÉNDICES

CARACTERÍSTICAS UTILIZADAS PARA EL MODELO

En la siguiente tabla se detallan las características extraídas por la herramienta CICFlowmeter, presentada en apartado 3.1.1.

Feature	Explicación	Feature	Explicación
ACK Flag Cnt	Número de paquetes con ACK.	Active Max	Tiempo máximo de un paquete antes de entrar IDLE.
Active Mean	Tiempo medio activo.	Active Std	Desviación típica sobre el tiempo activo.
Bwd Header Len	Tamaño del <i>header</i> entrante en bytes.	Bwd IAT Max	Tiempo máximo entre dos paquetes entrantes.
Bwd IAT Mean	Tiempo medio entre dos paquetes entrantes.	Bwd IAT Min	Tiempo mínimo entre dos paquetes entrantes.
Bwd IAT Std	Desviación típica del tiempo entre dos paquetes entrantes.	Bwd IAT Tot	Tiempo total entre dos paquetes entrantes.
Bwd Pkt Len Max	Tamaño máximo de un paquete entrante.	Bwd Pkt Len Mean	Tamaño medio de un paquete entrante.
Bwd Pkt Len Min	Tamaño mínimo de un paquete entrante.	Bwd Pkt Len Std	Desviación típica en el tamaño de paquetes entrantes.
Bwd Pkts/s	Paquetes entrantes por segundo.	Bwd Seg Size Avg	Tamaño medio observado en paquetes entrantes.
CWE Flag Count	Número de paquetes con flag CWE.	Down/Up Ratio	<i>Download/Upload Ratio</i>
ECE Flag Count	Número de paquetes con flag ECE.	FIN Flag Cnt	Número de paquetes con flag FIN.

Flow Duration	Duración del <i>flow</i> .	Flow IAT Max	Tiempo máximo entre dos <i>flows</i> .
Flow IAT Mean	Tiempo medio entre dos <i>flows</i> .	Flow IAT Min	Tiempo mínimo entre dos <i>flows</i> .
Flow IAT Std	Desviación típica en el tiempo entre <i>flows</i> .	Fwd Act Data Pkts	Número de paquetes con al menos 1 byte TCP.
Fwd Header Len	Tamaño del <i>header</i> saliente en bytes.	Fwd IAT Max	Tiempo máximo entre dos paquetes salientes.
Fwd IAT Mean	Tiempo medio entre dos paquetes salientes.	Fwd IAT Min	Tiempo mínimo entre dos paquetes salientes.
Fwd IAT Std	Desviación típica en el tiempo entre paquetes salientes.	Fwd IAT Tot	Tiempo total entre dos paquetes salientes.
Fwd PSH Flags	Número de veces que se activa PSH en paquetes salientes.	Fwd Pkt Len Max	Tamaño máximo de un paquete saliente.
Fwd Pkt Len Mean	Tamaño medio de un paquete saliente.	Fwd Pkt Len Min	Tamaño mínimo de un paquete saliente.
Fwd Pkt Len Std	Desviación típica del tamaño de paquetes salientes.	Fwd Pkts/s	Paquetes salientes por segundo.
Fwd Seg Size Avg	Tamaño medio observado en paquetes salientes	Fwd Seg Size Min	Tamaño mínimo observado en paquetes salientes
Fwd URG Flags	Número de veces que se activa URG en paquetes salientes.	Idle Max	Máximo tiempo transcurrido antes de que el <i>flow</i> se vuelva inactivo.
Idle Mean	Tiempo medio transcurrido antes de que el <i>flow</i> se vuelva inactivo.	Idle Min	Tiempo mínimo transcurrido antes de que el <i>flow</i> se vuelva inactivo.
Idle Std	Desviación típica del tiempo transcurrido antes de que el <i>flow</i> se vuelva inactivo.	Init Bwd Win Byts	Bytes recibidos en la ventana inicial.
Init Fwd Win Byts	Bytes enviados en la ventana inicial.	PSH Flag Cnt	Número de paquetes con PSH.

Pkt Len Max	Longitud máxima de un <i>flow</i> .	Pkt Len Mean	Longitud media de un <i>flow</i> .
Pkt Len Min	Longitud mínima de un <i>flow</i> .	Pkt Len Std	Desviación típica en la longitud de un <i>flow</i> .
Pkt Len Var	Tiempo mínimo entre llegadas de paquetes.	Pkt Size Avg	Tamaño medio de paquetes.
RST Flag Cnt	Número de paquetes con flag RST.	Subflow Bwd Byts	Media de bytes recibidos en <i>subflows</i> .
Subflow Bwd Pkts	Media de paquetes recibidos en <i>subflows</i> .	Subflow Fwd Byts	Media de bytes enviados en <i>subflows</i> .
Subflow Fwd Pkts	Media de paquetes enviados en <i>subflows</i> .	Tot Bwd Pkts	Total de paquetes recibidos en <i>subflows</i> .
Tot Fwd Pkts	Total de paquetes enviados en <i>subflows</i> .	TotLen Bwd Pkts	Longitud total de paquetes recibidos en <i>subflows</i> .
TotLen Fwd Pkts	Longitud total de paquetes enviados en <i>subflows</i> .	URG Flag Cnt	Número de paquetes con flag URG.

Tabla A.1: Lista de features extraídos por CICFlowmeter.

REQUISITOS DE LA APLICACIÓN

En este apéndice se presentan los distintos requisitos especificados en la apartado 5.2.

B.1. Requisitos Funcionales

RF-01: Creación de un nuevo usuario	
Actor	Usuario
Descripción	El sistema debe permitir la creación de un nuevo usuario al correrlo por primera vez.
Precondiciones	El Usuario ha iniciado la aplicación. No debe haber ningún usuario creado.
Postcondiciones	Se almacena en BBDD los datos del nuevo usuario. El Sistema solicita un inicio de sesión con las nuevas credenciales.
Escenario principal	
<ol style="list-style-type: none">1. El Usuario inicia la aplicación web.2. El Sistema muestra una página similar a la de inicio de sesión avisando de que no existe ningún usuario creado.3. El Usuario inserta las credenciales deseadas <i>username:password</i>.4. El Sistema verifica que se han insertado correctamente las credenciales y las almacena en BBDD.5. El Sistema solicita a continuación un inicio de sesión.	
Flujos Alternativos	
Ninguno	

Figura B.1: Requisito Funcional 1: Creación de un nuevo usuario.

RF-02: Inicio de sesión	
Actor	Usuario
Descripción	El sistema debe permitir el inicio de sesión.
Precondiciones	El Usuario ha iniciado la aplicación. Ya existe un usuario en la BBDD.
Postcondiciones	Se genera un nuevo token de sesión único en BBDD. El Sistema inicia una sesión y redirige al Usuario a la página principal de la aplicación.
Escenario principal	
<div>1. El Usuario inicia la aplicación web.</div> <div>2. El Sistema muestra un formulario de inicio de sesión.</div> <div>3. El Sistema comprueba que las credenciales introducidas coinciden con las de la BBDD.</div> <div>4a. Si las credenciales son correctas, el Sistema redirige al usuario a la página principal del proyecto, iniciando una sesión.</div> <div>4b. Si las credenciales son erróneas, se vuelve a mostrar el formulario junto con un mensaje de error.</div>	
Flujos Alternativos	
<div>1. No existe ningún usuario en BBDD.</div> <div>2. Se crea el Usuario tal y como se indica en RF-01.</div> <div>3. El Sistema crea un usuario en BBDD.</div> <div>4. Se muestra el formulario de inicio de sesión, y se procede como en el escenario principal.</div>	

Figura B.2: Requisito Funcional 2: Inicio de sesión.

RF-03: Iniciar/Pausar captura de tráfico	
Actor	Usuario
Descripción	El Sistema debe permitir iniciar/pausar una captura de tráfico en una interfaz concreta.
Precondiciones	El Usuario ha iniciado sesión.
Postcondiciones	Se inicia una captura de sesión y comienzan a mostrarse los resultados por pantalla.
Escenario principal	
<ol style="list-style-type: none"> 1. El Usuario pulsa en el botón "Launch". 2. Aparece un desplegable con las interfaces de red disponibles en el host. 3. El Usuario selecciona una de ellas. 4. El Sistema comienza a capturar tráfico de forma constante y a mostrar resultados por pantalla. 5. El botón "Launch" ha cambiado a "Stop". Si el Usuario pulsa el botón, el Sistema detiene la captura. 	
Flujos Alternativos	
Ninguno	

Figura B.3: Requisito Funcional 3: Iniciar/Pausar captura de tráfico.

RF-04: Filtrar datos en la tabla de flows	
Actor	Usuario
Descripción	El Sistema debe permitir filtrar y ordenar datos en la tabla que muestra los flows capturados.
Precondiciones	El Usuario ha iniciado sesión. El Sistema ha iniciado una captura de tráfico y ha obtenido algunos flows.
Postcondiciones	Se muestran por pantalla los datos filtrados u ordenados.
Escenario principal	
<ol style="list-style-type: none"> 1. El Sistema ha iniciado previamente una captura de tráfico y se están mostrando por pantalla los resultados. 2. El Usuario puede escribir en una barra de búsqueda una palabra con la que filtrar datos de la tabla. 3. El Usuario puede seleccionar una columna y ordenar su contenido de forma ascendente/descendente. 	
Flujos Alternativos	
Ninguno	

Figura B.4: Requisito Funcional 4: Filtrar datos en la tabla de flows.

RF-05: Acceder a estadísticas del tráfico	
Actor	Usuario
Descripción	El Sistema debe mostrar estadísticas del tráfico en la red.
Precondiciones	El Usuario ha iniciado sesión. El Sistema ha iniciado una captura de tráfico y ha obtenido algunos flows.
Postcondiciones	Se muestran por pantalla gráficas que reflejan información del tráfico en la red.
Escenario principal	
<div>1. El Sistema ha iniciado previamente una captura de tráfico y se están mostrando por pantalla los resultados.</div> <div>2. El Usuario puede escribir en una barra de búsqueda una palabra con la que filtrar datos de la tabla.</div> <div>3. El Usuario puede seleccionar una columna y ordenar su contenido de forma ascendente/descendente.</div>	
Flujos Alternativos	
Ninguno	

Figura B.5: Requisito Funcional 5: Acceder a estadísticas del tráfico.

RF-06: Fijar una IP como objetivo	
Actor	Usuario
Descripción	El Sistema debe permitir fijar una IP como objetivo, para acceder a información concreta.
Precondiciones	El Usuario ha iniciado sesión. El Sistema ha iniciado una captura de tráfico y ha obtenido algunos flows.
Postcondiciones	Se muestran por pantalla gráficas y datos que reflejan información de una IP concreta.
Escenario principal	
<ol style="list-style-type: none"> 1. El Sistema ha iniciado previamente una captura de tráfico y se están mostrando por pantalla los resultados. 2. El Usuario pulsa en un flow en el que aparezca la IP deseada. 3. El Usuario pulsa el botón "Set Target". 4. El Sistema lo añade a la lista de Objetivos y a la BBDD. 5. El Usuario puede ahora consultar la información específica en la pestaña Objetivos. 	
Flujos Alternativos	
Ninguno	

Figura B.6: Requisito Funcional 6: Fijar una IP como objetivo.

RF-07: Bloquear una IP	
Actor	Usuario
Descripción	El Sistema debe permitir bloquear una IP que haya sido fijada como objetivo.
Precondiciones	El Usuario ha iniciado sesión. El Sistema ha iniciado una captura de tráfico y ha obtenido algunos flows. La IP a bloquear se ha fijado como objetivo y no ha sido bloqueada previamente.
Postcondiciones	El dispositivo con la IP seleccionada queda bloqueado en la red. Se incluye en la BBDD que la IP ha sido bloqueada. El objetivo aparece como bloqueado en la pestaña de Objetivos.
Escenario principal	
1. El Usuario visita la pestaña de Objetivos. 2. Pulsa en el botón "Bloquear" sobre uno de los objetivos. 3a. Si se ha bloqueado con éxito, el Sistema muestra un mensaje y cambia el botón a "Desbloquear". 3b. Si la IP ya ha sido bloqueada, se notifica y no ocurre nada más. 3c. Si ocurre algún error, el Sistema lo notifica.	
Flujos Alternativos	
Ninguno	

Figura B.7: Requisito Funcional 7: Bloquear una IP.

RF-08: Desbloquear una IP	
Actor	Usuario
Descripción	El Sistema debe permitir desbloquear una IP que haya sido fijada como objetivo.
Precondiciones	El Usuario ha iniciado sesión. El Sistema ha iniciado una captura de tráfico y ha obtenido algunos flows. La IP deseada se ha bloqueado previamente.
Postcondiciones	La IP queda desbloqueada en la red. Se incluye en la BBDD que la IP ha sido desbloqueada. El objetivo aparece como desbloqueado en la pestaña de Objetivos.
Escenario principal	
1. El Usuario visita la pestaña de Objetivos. 2. Pulsa en el botón "Desbloquear" sobre uno de los objetivos. 3a. Si se ha desbloqueado con éxito, el Sistema muestra un mensaje y cambia el botón a "Bloquear". 3b. Si ocurre algún error, el Sistema lo notifica.	
Flujos Alternativos	
Ninguno	

Figura B.8: Requisito Funcional 8: Desbloquear una IP.

RF-09: Cierre de sesión	
Actor	Usuario
Descripción	El Sistema debe permitir que un usuario que ha iniciado sesión, la finalice.
Precondiciones	El Usuario ha iniciado sesión.
Postcondiciones	El Sistema cierra la sesión y redirige al Usuario a la página de inicio de sesión. Ya no se puede acceder a ninguna funcionalidad interna de la aplicación.
Escenario principal	
<ol style="list-style-type: none">1. El Usuario pulsa el botón "Logout".2. El Sistema elimina todas las cookies y elimina la sesión en el servidor.3. Se redirige al Usuario a la página de inicio de sesión.	
Flujos Alternativos	
Ninguno	

Figura B.9: Requisito Funcional 9: Cerrar sesión.

TECNOLOGÍA EMPLEADA

Debido, sobre todo, a la diversidad de los módulos a implementar, fue necesario plantear el uso de un gran abanico de tecnologías y lenguajes de programación para conseguir un buen resultado.

A continuación, se incluye una descripción de las diversas tecnologías empleadas y los motivos por los que se eligieron.

C.1. NodeJS



Se trata de un entorno en tiempo de ejecución basado en JavaScript, ideado principalmente para su implantación a nivel de servidor. Está basado en el motor de alto rendimiento de código abierto V8 [9], diseñado por Google.

Permite crear programas con una alta escalabilidad y, gracias a su modelo mono-hilo pensado para tareas asíncronas, resulta ideal para gestionar el backend de una API REST.

Además, cuenta con una extensa comunidad y un inmenso catálogo de paquetes que facilitan enormemente la tarea del desarrollador.

C.2. VueJS



La programación reactiva es una tendencia indudable dentro del desarrollo web. Entre los frameworks reactivos más utilizados se encuentra VueJS, cuyos desarrolladores lo definen como «un framework progresivo para construir interfaces de usuario», lo que implica que cuenta con una gran cantidad de funcionalidades distintas que podemos ir incluyendo según las vamos necesitando.

Su gran modularización, su facilidad de uso y su activa comunidad hacen de esta herramienta el framework ideal para construir nuestra interfaz web.

C.3. Python



Uno de los lenguajes de programación más utilizados hoy en día, muy fácil de usar y con una inmensa cantidad de librerías. Python es, probablemente, el lenguaje de programación más versátil que existe.

Sin embargo, la decisión de utilizar Python dentro de nuestra aplicación fue tomada en base a la necesidad de incluir la librería *sklearn* para trabajar con el modelo de predicción de ataques.

C.4. MySQL



MySQL se encuentra entre los sistemas de gestión de bases de datos más conocidos en la actualidad. Pese a que se trata de una aplicación de código cerrado, su facilidad de instalación y de uso nos llevaron a incluir este gestor en nuestra aplicación.

C.5. Sequelize



Ya mencionamos que NodeJS cuenta con una inmensa cantidad de paquetes para prácticamente todas las necesidades. En concreto, Sequelize es la herramienta ideal para gestionar la interacción con la base de datos desde un servidor. Se trata de un Object-Relational Mapping (ORM) que permite convertir tablas de una base de datos en entidades accesibles desde Node, facilitando inmensamente la interacción con los datos almacenados.

C.6. Docker



Con el objetivo de llevar la modularización de nuestra aplicación hasta el máximo nivel y facilitar su despliegue, decidimos enfocar el desarrollo hacia el uso de contenedores. Un contenedor proporciona virtualización a nivel de sistema operativo, lo que permite que existan múltiples instancias aisladas de espacios de usuario.

Docker es un proyecto de código abierto que permite automatizar el despliegue de este tipo de contenedores. Su sencillez, así como su extensa comunidad, lo convirtieron en nuestra elección para esta aplicación.



ESTRUCTURA DE FICHEROS

La estructura de ficheros está íntimamente relacionada con el despliegue de los diversos contenedores. A continuación se detallan los diferentes directorios y su contenido.

Directorio Raíz. Aquí se incluyen los ficheros de instalación y el archivo *docker-compose.yml*, necesario para el despliegue de contenedores.

./NIDS/ Cuenta con un fichero *Dockerfile*, con las instrucciones de despliegue del contenedor y el fichero *package.json* que contiene las dependencias de la aplicación.

NIDS/config/ Incluye ficheros de configuración, credenciales y variables de entorno.

NIDS/src/ Recoge todo el código de este módulo. El fichero *server.js* es el del propio servidor y su configuración.

NIDS/src/core/core.js Incluye la lógica de la API. Cada función representa una posible respuesta. Actúa como intermediario entre la API y los submódulos.

NIDS/src/routes/routes.js Declaración de las URIs de la API. Enlaza cada petición con una función de *core.js*.

NIDS/src/database/ Incluye las librerías de interacción con el ORM, una por cada clase. El fichero *csv2database.js* es el encargado de introducir los *flows* capturados en la base de datos.

NIDS/src/certs/ Directorio para almacenar los certificados SSL.

NIDS/src/ML/ Recoge la funcionalidad de predicción de ataques en Python.

Los directorios **flowmeter/**, y **sniffer/** incluyen la funcionalidad de captura de tráfico y transformación a CSV.

./frontend/. Su estructura es muy similar a la del NIDS. El directorio **public/** incluye todos los archivos estáticos de la interfaz web, y **views/** incluye la plantilla para el inicio de sesión.

./docker_config/ Incluye el fichero de creación de la base de datos y algunos archivos de configuración.

MÉTODOS DE LA API

E.1. Métodos de la API NIDS

Se trata de la API principal, alojada en el contenedor NIDS, puerto 8080. Desde el exterior, es accesible usando la URL **https://<IP del host>:8080/**. El cuadro E.1 incluye los métodos de la API NIDS y una descripción de cada uno.

URI	Método	Descripción
/api/sniffer/start	POST	Lanza el <i>sniffer</i> .
/api/sniffer/stop	POST	Detiene el <i>sniffer</i> .
/api/sniffer/reset	POST	Elimina los ficheros temporales.
/api/sniffer/isRunning	GET	Comprueba si el <i>sniffer</i> está corriendo.
/api/sniffer/getInterfaces	GET	Devuelve las interfaces de red disponibles.
/api/ddbb/flows/destroyAll	POST	Elimina todos los <i>flows</i> de la base de datos.
/api/ddbb/flows/getFromHour	GET	Devuelve un array con todos los <i>flows</i> capturados entre la hora especificada y la siguiente.
/api/ddbb/flows/getCurrentHour	GET	Devuelve un array con todos los <i>flows</i> de la última hora.
/api/ddbb/flows/getChartTrafficTime	GET	Devuelve un array de <i>flows</i> junto con su <i>timestamp</i> y su longitud para mostrar en la tabla.
/api/ddbb/flows/getIPTrafficData	GET	Devuelve un array de <i>flows</i> junto con su longitud en Mb.
/api/ddbb/flows/getAttacksIPData	GET	Devuelve un array de <i>flows</i> junto con su proporción de ataques/benignos.
/api/ddbb/ips/setTarget	POST	Añade un objetivo a la BBDD.
/api/ddbb/ips/removeTarget	POST	Elimina un objetivo de la BBDD (a no ser que esté bloqueado).
/api/ddbb/ips/getTargets	GET	Devuelve todos los objetivos.

/api/ddbb/ips/isTargeted	POST	Comprueba si una IP está marcada como objetivo.
/api/ddbb/ips/getAttacksFromIP	POST	Devuelve el Top 10 de IPs más atacadas por un objetivo en orden descendente.
/api/ddbb/ips/block	POST	Bloquea un objetivo.
/api/ddbb/ips/unblock	POST	Desbloquea un objetivo.
/api/users/newUser	POST	Crea un nuevo usuario en BBDD.
/api/users/emptyUsers	POST	Comprueba si existen usuarios en BBDD.
/api/users/checkLogin	POST	Comprueba las credenciales de acceso.

Tabla E.1: Lista de métodos de la API NIDS.

E.2. Métodos de la API Frontend

Esta API es bastante más sencilla. Se accede desde **https://<IP del host>:80000/**. En el cuadro E.2 se detallan sus métodos.

URI	Método	Descripción
/	GET	Muestra la página principal o la de login si no hay sesión iniciada.
/login	POST	Comprueba las credenciales de acceso.
/logout	GET	Cierra sesión y elimina las cookies.
/newUser	POST	Envía las credenciales para la creación de un nuevo usuario a microservicio del NIDS. Si se crea correctamente, redirige a /.

Tabla E.2: Lista de métodos de la API Frontend.

EJEMPLOS DE ESCALABILIDAD DE LA APLICACIÓN

F.1. Utilizar un nuevo modelo de predicción

Ya mencionamos en el capítulo 3 que el modelo que usemos dependerá de la infraestructura en la que se vaya a desplegar. Por lo tanto, será necesario entrenar un modelo previamente en el entorno en el que se desee utilizar.

Con este objetivo se incluye en el repositorio un *notebook* `train.ipynb`, capaz de trabajar sobre un *dataset* dado. El resultado será un *pickle* que habrá que guardar como `./NIDS/src/ML/pickles/LogReg_1.pkl`.

Es crucial que el *dataset* que empleemos sea el resultado de utilizar la versión adaptada de CIC-FlowMeter, cuyo ejecutable se encuentra en `./NIDS/src/flowmeter/bin/cfm`. Una vez tengamos el archivo `pcap`, será necesario situarse en el directorio que acabamos de mencionar y ejecutar el siguiente comando:

```
./cfm <ruta relativa del fichero .pcap><directorio de salida>
```

El *notebook* está pensado para ejecutarse en Google Colab, por lo que si se desea utilizar un entorno local, como Jupyter, será necesario adaptarlo ligeramente.

F.2. Añadir una nueva gráfica al Frontend

Supongamos que queremos añadir una nueva gráfica al Frontend que, además, muestre unas estadísticas que hasta ahora no calculaba el Backend. En ese caso, será necesario modificar varios ficheros:

./NIDS/src/core/core.js Aquí se añadirá la función que gestione la petición de la API. Primero habrá que comprobar que la información recibida (por ejemplo, el token de sesión) es válida y, a continuación, se realizará una llamada a base de datos para obtener la información

necesaria, realizar los cálculos y responder con el resultado. En caso de que no exista la función del ORM que nos devuelva tal información, se debe crear en **./NIDS/src/database/flows.js**.

./NIDS/src/routes/routes.js Aquí se añadirá una entrada que enlace la URI deseada con la función que acabamos de crear en **core.js**.

./frontend/src/public/js/app.js Aquí será necesario crear un nuevo componente Vue que realice la petición al Backend, gestione la respuesta y muestre el resultado por pantalla.

Se trata de un ejemplo general, por lo que es muy probable que no baste con modificar estos archivos, aunque eso ya depende exclusivamente de la funcionalidad que queramos añadir.

SEGURIDAD

En este apéndice se detallan los diferentes criterios de seguridad seguidos durante el desarrollo de la aplicación.

G.1. Seguridad en Docker

En un primer momento puede parecer que el hecho de que el servidor esté desplegado dentro de un contenedor, aislado del resto del sistema y con un espacio de ejecución independiente, es motivo suficiente para descuidar la seguridad dentro del propio contenedor. Sin embargo, una mala gestión del servicio puede llegar a permitir que un atacante sea capaz de ejecutar código en el servidor e incluso escalar privilegios hasta acceder a recursos externos del host [10].

En nuestro caso, como la única forma de acceder al contenedor es a través de la API, es necesario prestar especial atención a las posibles vulnerabilidades de inyección de código. Es por eso que todos los parámetros de entrada a la API se filtran para evitar cadenas maliciosas.

G.2. Tráfico cifrado

Un ataque bastante común en el ámbito de las redes es el Man-In-The-Middle (MITM), que consiste en interceptar el tráfico entre dos *endpoints*, teniendo acceso a toda la información intercambiada. Una forma de mitigar el riesgo de estos ataques es cifrando la comunicación entre ambos puntos.

Para ello hemos decidido instalar certificados Secure Sockets Layer (SSL) tanto en el servidor NIDS como en el de Frontend, de modo que tanto la web como la API establecen comunicaciones mediante el protocolo HTTPS. En la figura G.1 se puede ver que los algoritmos utilizados para proteger la comunicación son *AES-256 GCM* (cifrado simétrico con autenticación) y *SHA-384* (algoritmo de *hashing* para asegurar la integridad).

Estos certificados se generaron con OpenSSL en el propio servidor, por lo que no cuentan con la aceptación de una Autoridad Certificadora. La mayoría de navegadores mostrarán una alerta al

Technical Details

Connection Encrypted (TLS_AES_256_GCM_SHA384, 256 bit keys, TLS 1.3)

The page you are viewing was encrypted before being transmitted over the Internet.

Figura G.1: Algoritmo establecido por el certificado SSL.

acceder a la web, por lo que deberemos añadir una excepción para poder utilizarla.

G.3. Usuario único

Por motivos de seguridad, únicamente se permite la creación de un usuario, que corresponderá al administrador del sistema. Esto podrá hacerse exclusivamente al iniciar la aplicación por primera vez. El sistema comprueba que no existe ningún usuario creado y muestra una pantalla de inicio de sesión con un mensaje que alerta de que las credenciales introducidas serán las que se utilicen a partir de ese momento.

G.4. Seguridad en la API

Los propios métodos de la API permiten acceder a información sensible e incluso bloquear dispositivos en la red. Por lo tanto, es necesario limitar su uso únicamente a aquellos usuarios autenticados.

En un primer momento se pensó utilizar el software Keycloak [11]. Se trata de una solución muy completa que permite añadir autenticación y gestión de identidades. Sin embargo, Keycloak es un software muy completo y complejo y, puesto que lo que buscábamos realmente era muy sencillo, se decidió diseñar una solución propia basada en el uso de *bearer tokens* (cadenas de texto únicas y aleatorias que sirven para demostrar que el usuario que realiza una petición se ha autenticado previamente).

Los requisitos necesarios son los siguientes:

- Únicamente usuarios registrados en la plataforma pueden hacer peticiones a la API.
- Únicamente puede haber un usuario conectado a la vez.
- El *token* de sesión debe ser aleatorio y lo suficientemente largo para que no pueda falsificarse.
- El *token* de sesión debe ser único y no puede reutilizarse.

En nuestra solución, cada vez que un usuario solicita un inicio de sesión con unas credenciales, el sistema comprueba que estas son válidas y, en caso afirmativo, genera un *token* de 64 B con la función `crypto.randomBytes(64).toString('hex')` y se actualiza el campo `token` del usuario en

la base de datos.

En todo momento, los métodos de la API NIDS esperan un parámetro `token` y un parámetro `username`. En caso de que no existan o el `token` no sea correcto, responde con un `status: 401` y genera un error de autenticación.

De esta forma evitamos reciclar `tokens` y, además, garantizamos que no pueda haber dos usuarios accediendo a la API de forma simultánea, puesto que al acceder con las credenciales de administrador se genera un nuevo token, invalidando todos los que estuvieran siendo usados.

Para gestionar las sesiones, se utiliza el paquete de NodeJS `express-session`, que permite gestionar sesiones cifradas. Decidimos incluir tres parámetros en la sesión: `username`, `token` y `status`. De esta forma, cada vez que se recargue la página el servidor sabrá si existe alguna sesión activa.

Sin embargo, las sesiones generadas por este paquete únicamente son accesibles desde el servidor. Por lo tanto, decidimos añadir también dos `cookies`: una con el `token` y otra con el `username`, para que el frontend pudiera realizar peticiones a la API NIDS. En todo momento se comprueba que los valores de las `cookies` coinciden con los de la sesión y, además, cada vez que se carga la página se comprueba que el `token` sigue siendo válido, realizando una petición de prueba contra la API NIDS.

En la figura G.2 se incluye un diagrama de secuencia que refleja el proceso de obtención de un `token`.

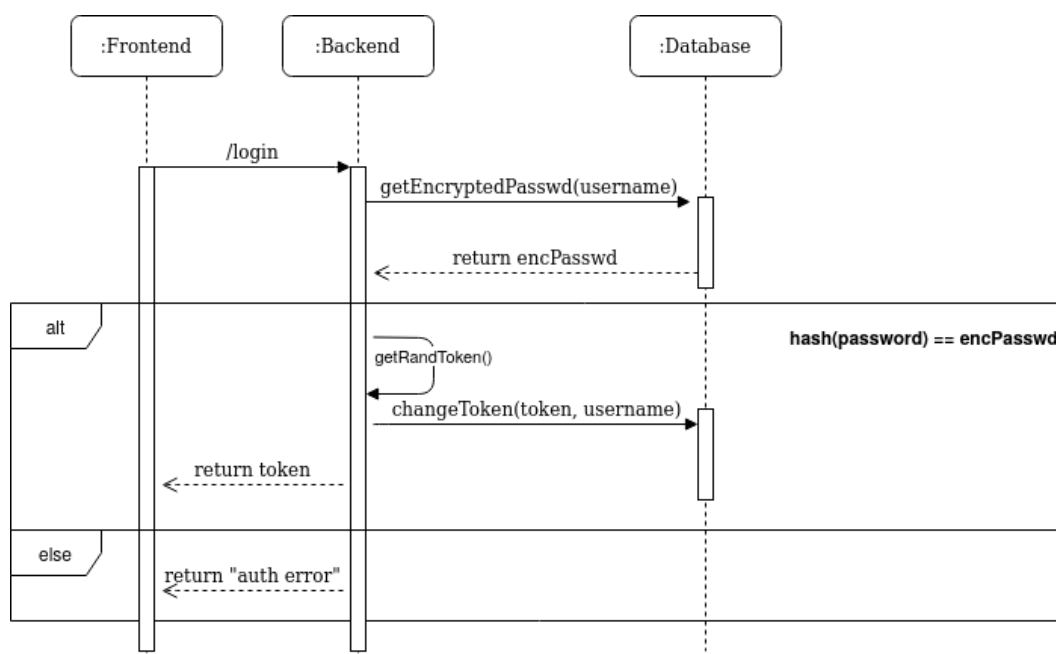


Figura G.2: Diagrama de secuencia para la obtención de un token.

G.5. Gestión segura de contraseñas

Se evita en todo momento almacenar contraseñas en texto plano en la base de datos. En su lugar, emplearemos el algoritmo `bcrypt` para obtener un *hash* de la contraseña, que será lo que almacenemos, tal y como se recomienda en la guía de OWASP para gestión de contraseñas [12].

`bcrypt` es una función para generar *hashes* de contraseñas basado en el algoritmo `Blowfish`. Incorpora un fragmento de texto aleatorio a la contraseña, denominado `salt`, para hacerlo seguro frente a ataques que usen tablas *rainbow* (tablas de *hashes* precomputados empleadas para *crackear* funciones *hash*). Se trata, además, de uno de los algoritmos más extendidos para gestión de contraseñas y cuenta con implementaciones para un amplio abanico de lenguajes.

Para generar un *hash*, basta llamar a la función `bcrypt.hashSync(password, 12)`, donde 12 es el coste del algoritmo. El coste se calcula como $\log_2(\text{iteraciones})$ por lo que, en nuestro caso, el algoritmo tendrá 2^{12} iteraciones, que es lo recomendado por OWASP.

El *hash* obtenido por `bcrypt` tiene la siguiente forma

```
$2b$12$Nz6XmG3pdW5REecXQhauQOu/zxSMGphHxuVFGhe.Qjx6oIR/yBd9a
```

donde el prefijo `$2b$` indica que la cadena es un *hash* `bcrypt` en su última versión, mientras que el prefijo `12$` indica el coste empleado. A continuación se incluyen 22 caracteres que corresponden con el `salt`, seguidos de los 31 caracteres del *hash*, codificados usando `Radix-64` (una variación de `Base-64`).

G.6. Bloqueo de IPs

Una de las funcionalidades más importantes de VISION es que permite bloquear una dirección IP en la red desde el panel de control, lo que se consigue añadiendo una regla en `IPTables` (utilidad de Linux que permite gestionar el cortafuegos del sistema). Sin embargo, para hacer esto posible es necesario establecer un canal de comunicación entre el contenedor NIDS y el host, para que el proceso de NIDS pueda ejecutar el comando `iptables` desde el propio host (ver apéndice I).

Para conseguir esto de forma segura, hemos optado por utilizar el protocolo SSH. Cuando el servidor NIDS recibe la orden de bloquear una IP y verifica que el usuario que la emite se ha autenticado correctamente, se ejecuta un comando SSH desde dentro del contenedor hacia el host, con las credenciales de un usuario de la máquina que se encuentran en el archivo `NIDS/config/ssh-credential.js`.

Resulta crucial que el usuario tenga los mínimos privilegios y que exista una entrada en el archivo `/etc/sudoers` que le permita ejecutar `iptables` (y nada más) como `root`. Un ejemplo de entrada para el archivo `/etc/sudoers` sería:

```
username ALL=(ALL) NOPASSWD: /sbin/iptables.
```

Una mala configuración en este punto resulta muy peligrosa, puesto que podría desembocar en una escalada de privilegios sobre el host.

Para ejecutar el comando SSH hemos evitado usar funciones como `exec()` que son fácilmente inyectables. En su lugar, hemos optado por el paquete `node-ssh`, ideado precisamente para situaciones así.



MANUAL DE USUARIO

En este apéndice se proporciona un manual para el usuario, con la intención de facilitar la comprensión de la aplicación y mostrar el proceso de despliegue.

H.1. Instalación y despliegue

VISION es una aplicación que combina múltiples tecnologías y cuenta con diversos módulos muy variados, por lo que tiene una larga lista de dependencias.

El uso de contenedores Docker simplifica en gran medida la instalación y el despliegue de la aplicación y, junto con el Node Packet Manager (NPM), se ha conseguido automatizar todo el proceso.

El único prerequisite para el despliegue de VISION es tener un sistema Linux con `docker` y `docker-compose` instalados. Bastará entonces con ejecutar los comandos

```
docker-compose build
docker-compose run
```

para desplegar la aplicación. Los contenedores se levantarán automáticamente e instalarán todas las dependencias software necesarias.

Recordamos aquí la necesidad de modificar el archivo `NIDS/config/ssh-credentials.js` e introducir las credenciales del usuario SSH. Además, es muy importante cambiar las credenciales de la base de datos en los dos archivos que se encuentran en `docker_config/dev`.

H.2. Manual de uso

En esta sección se incluyen algunos ejemplos de uso de la aplicación, con imágenes para ayudar al usuario a comprender el funcionamiento de VISION.

H.2.1. Primer inicio de sesión

Cuando iniciamos VISION por primera vez, es introducir las credenciales de acceso para el administrador. No habrá forma de cambiarlas más adelante desde la aplicación, por lo que habrá que introducirlas con cuidado. La figura H.1 muestra la ventana de creación de usuario.

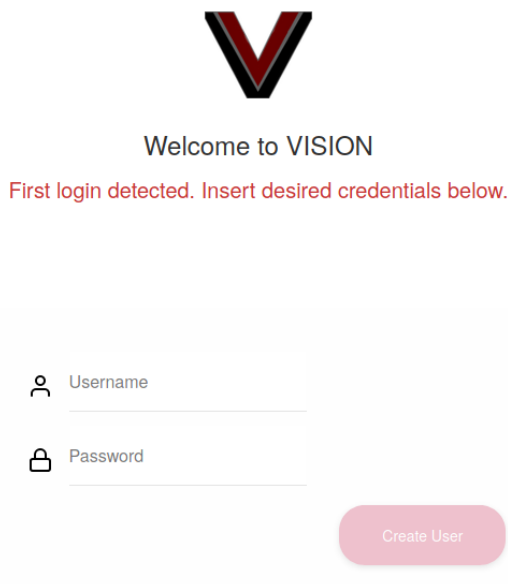


Figura H.1: Ventana de creación de usuario.

H.2.2. Inicio de sesión

Una vez existe un usuario en la base de datos, podremos iniciar sesión en la aplicación. La figura H.2 muestra el formulario de inicio de sesión.

H.2.3. Ventana principal

Al iniciar sesión se muestra la ventana principal de la aplicación. Desde aquí podremos acceder a las distintas pestañas de la aplicación (figura H.3), lanzar el *sniffer*, acceder a la documentación o cerrar sesión (figura H.4).

H.2.4. Lanzamiento del sniffer

Para comenzar a capturar tráfico, es necesario activar el *sniffer* seleccionando una de las interfaces disponibles en el host, tal y como se muestra en la figura H.5.



Welcome to VISION

A login form with two input fields: 'Username' and 'Password'. Each field has a corresponding icon (a person for username and a lock for password) to its left. To the right of the fields is a pink 'Log in' button.

Figura H.2: Ventana de inicio de sesión.



Figura H.3: Desde la ventana principal se puede acceder a las distintas pestañas de la aplicación.

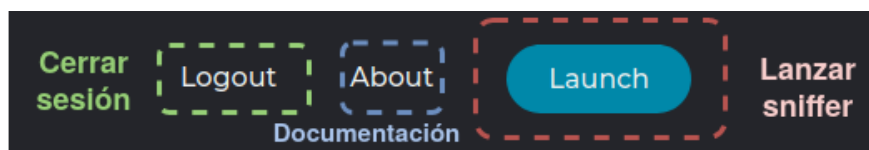


Figura H.4: Menú superior de la aplicación.

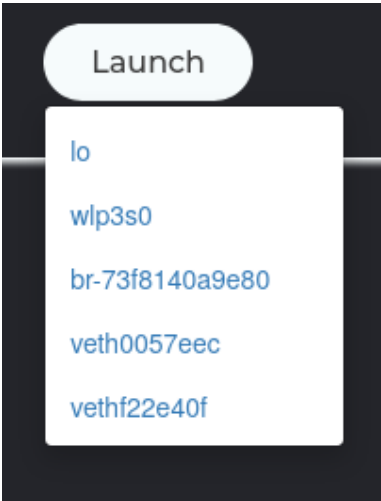


Figura H.5: Interfaces disponibles para comenzar a capturar tráfico.

H.2.5. Dashboard

Una vez ha comenzado la captura de tráfico, en el *dashboard* se muestran los *flows* capturados y si se han etiquetado como ataques o no. Desde aquí podremos filtrar los datos u ordenarlos como queramos. La figura H.6 muestra los diferentes componentes del *dashboard*.

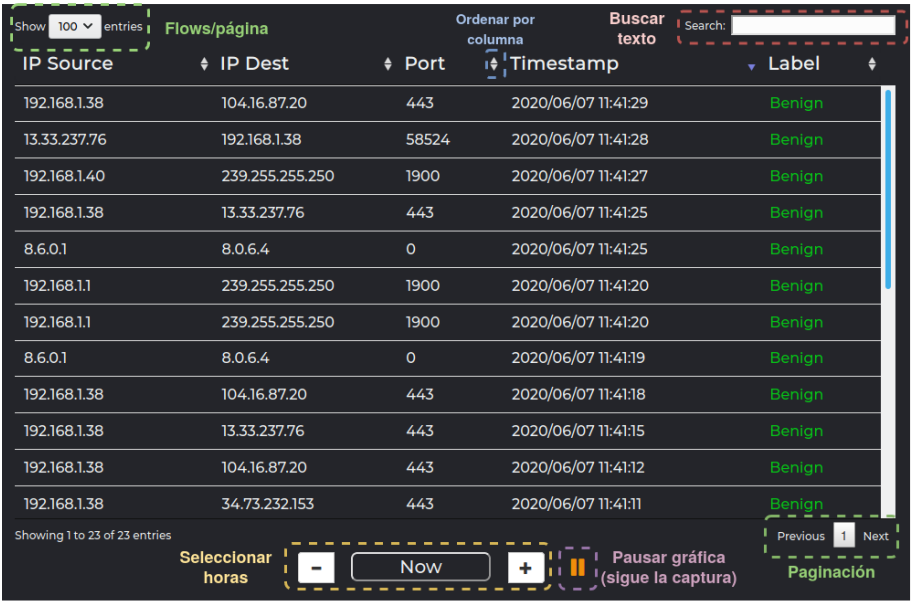


Figura H.6: Visualización de tráfico capturado en el *dashboard*.

H.2.6. Gráficas

Si seleccionamos esta pestaña, tendremos acceso a tres tipos distintos de gráficas. La primera muestra el flujo de tráfico por unidad de tiempo (en bytes). La segunda, muestra un *bar chart* con los bytes enviados por IP. La tercera refleja el porcentaje de paquetes enviados que se han etiquetado como ataques en cada IP. La figura H.7 muestra un ejemplo de gráfica del primer tipo.

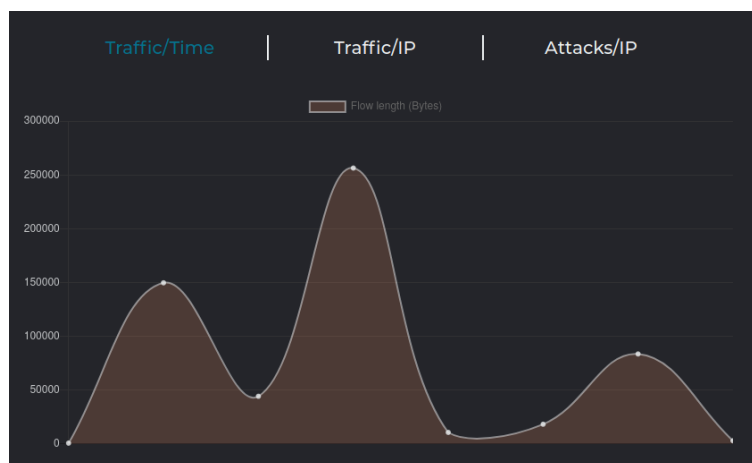


Figura H.7: Ejemplo de gráfica obtenida por la aplicación.

H.2.7. Objetivos

VISION permite fijar una serie de IPs objetivo, de modo que se pueda acceder a estadísticas específicas a cada una de ellas. Desde el *dashboard*, al pinchar sobre un *flow* se muestra una ventana desde la que podremos seleccionar una de las IPs implicadas en el flujo y fijarla como objetivo (figura H.8).

Si ahora accedemos a la pestaña de Objetivos, tendremos acceso a todas aquellas IPs que hayamos marcado. La figura H.9 muestra la tabla de los objetivos, desde donde podremos retirar la IP de la tabla, bloquear la IP en la red o, si pinchamos sobre la fila, acceder a una gráfica que muestra las IPs más atacadas por el objetivo (figura H.10).

La columna *% attacks last hour* refleja la proporción de ataques/recibidos etiquetados como ataques durante la última hora. La columna *Status* indica si el objetivo se encuentra actualmente bloqueado en la red o no.

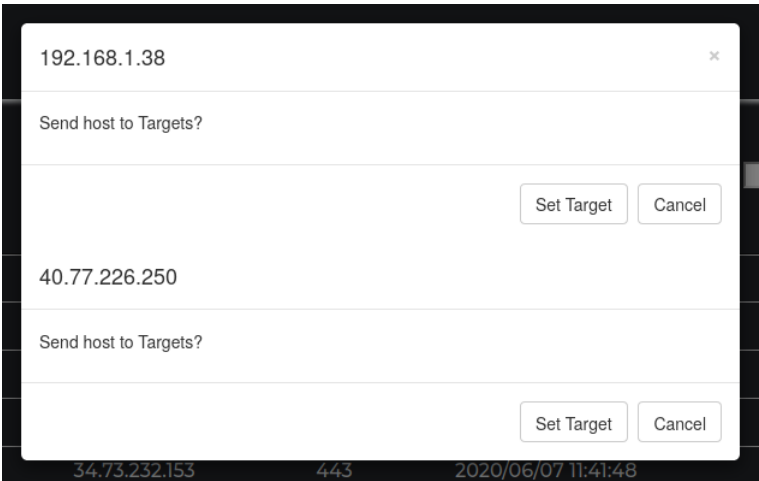


Figura H.8: Desde el *dashboard* se puede fijar un objetivo para su análisis detallado.





IP source	% attacks last hour	Status	Options
192.168.1.1	0.00 (src) / 0.00 (dst)	Online	  Bloquear IP
239.255.255.250	0.00 (src) / 0.00 (dst)	Online	 
Retirar objetivo			

Figura H.9: Tabla con los distintos objetivos fijados. Es posible retirar el objetivo, bloquear la IP o acceder a una gráfica de los ataques realizados.

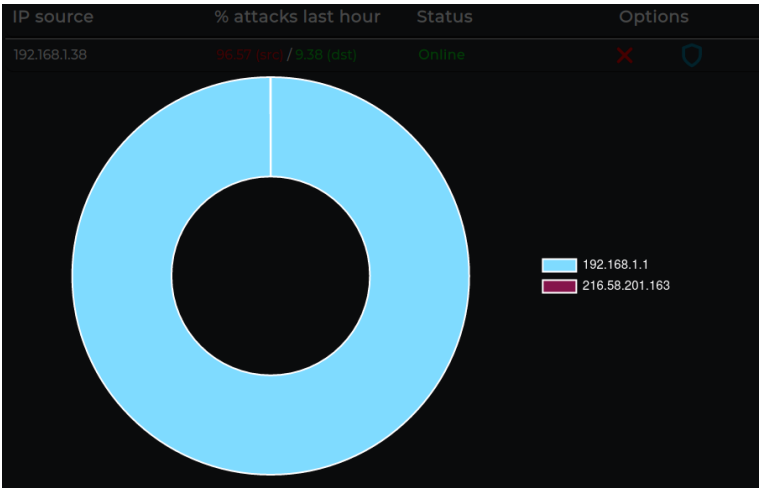


Figura H.10: Al pulsar sobre una IP objetivo aparece una gráfica que muestra las IPs más atacadas por el objetivo.

IPTABLES

Iptables es una utilidad de línea de comandos que permite configurar el *firewall* del kernel de Linux. Permite modificar, inspeccionar, redirigir o bloquear paquetes de red a nivel de la capa de red/IP.

I.1. Composición de Iptables

Iptables cuenta con una serie de tablas, cada una con una utilidad específica. Por defecto, la tabla que se utiliza es *filter*. Estas tablas contienen *cadenas*, que son listas de reglas que se aplican de forma secuencial y consecutiva.

Como se puede apreciar en el esquema, la tabla *filter* contiene tres cadenas: *INPUT*, *FORWARD* y *OUTPUT*. Estas cadenas integradas permiten aplicar las reglas antes, durante y después del proceso de enrutamiento.

Las reglas que componen una cadena permiten filtrar paquetes especificando *matches* (o «coincidencias») y *targets* (u «objetivos»). Las coincidencias indican sobre qué paquetes debe actuar la regla, mientras que los objetivos marcan las acciones a realizar una vez se haya dado una coincidencia.

Los objetivos integrados son *ACCEPT*, *QUEUE*, *DROP* y *RETURN* y se especifican mediante la opción *-j*.

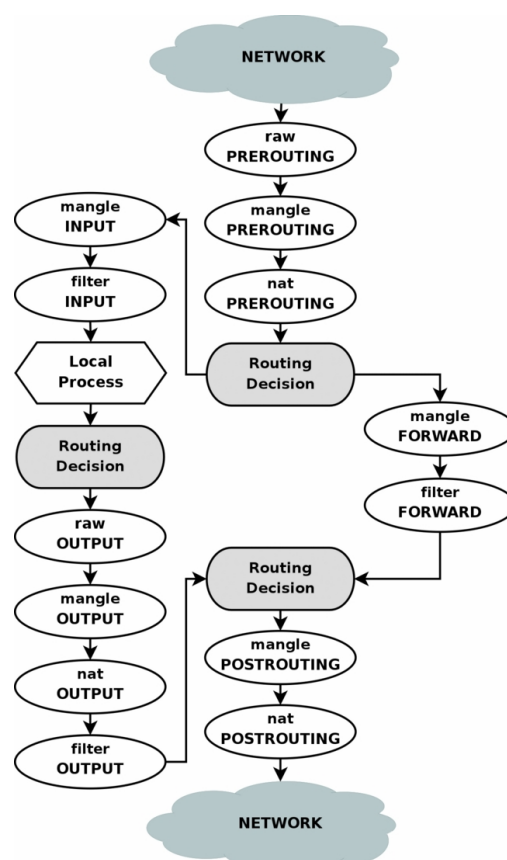


Figura I.1: Esquema de acción de las distintas tablas en *iptables*.

I.2. Generación de reglas

A la hora de generar una regla, debemos decidir, en primer lugar, si queremos limitar la regla a una interfaz de red concreta. Se puede especificar con las opciones `-i` (interfaz de entrada) y `-o` (interfaz de salida).

Debemos especificar la posición que queremos que ocupe nuestra regla en la cadena. La opción `-A` la incluirá al final de la cadena, y la opción `-I`, al comienzo.

A continuación, nos plantearemos en qué momento queremos interceptar el paquete. En nuestro caso, utilizaremos la cadena `FORWARD` puesto que actúa durante el proceso de enrutamiento.

El siguiente paso es elegir los *matches*. En nuestro caso, queremos filtrar los paquetes que procedan de la IP atacante, así que usaremos la opción `-s <IP>`. Si quisiéramos bloquear también los paquetes destinados a esa IP, usaríamos la opción `-d <IP>`.

Finalmente, elegimos el objetivo. Como queremos bloquear la IP atacante, usaremos la opción `-j DROP`.

Por lo tanto, la regla que hemos aplicado es la siguiente:

```
iptables -I FORWARD -s <IP>-j DROP
```

I.3. Limitaciones de la aplicación

Como ya se ha mencionado, *IPtables* es una herramienta que filtra paquetes a nivel de la capa de red. Esto impide que VISION pueda bloquear IPs que se encuentren en la misma subred, puesto que en ese caso el tráfico se enruta desde la capa de enlace (lo que se conoce como *switching*). Esto hace que, dentro de una misma subre, VISION sea capaz de detectar un ataque, pero no de bloquearlo.

Por lo tanto, actualmente es necesario que alguna de las IPs implicadas se encuentren fuera de la subred para que el enrutamiento de paquetes se produzca a nivel de red e *IPtables* pueda bloquearlos.

